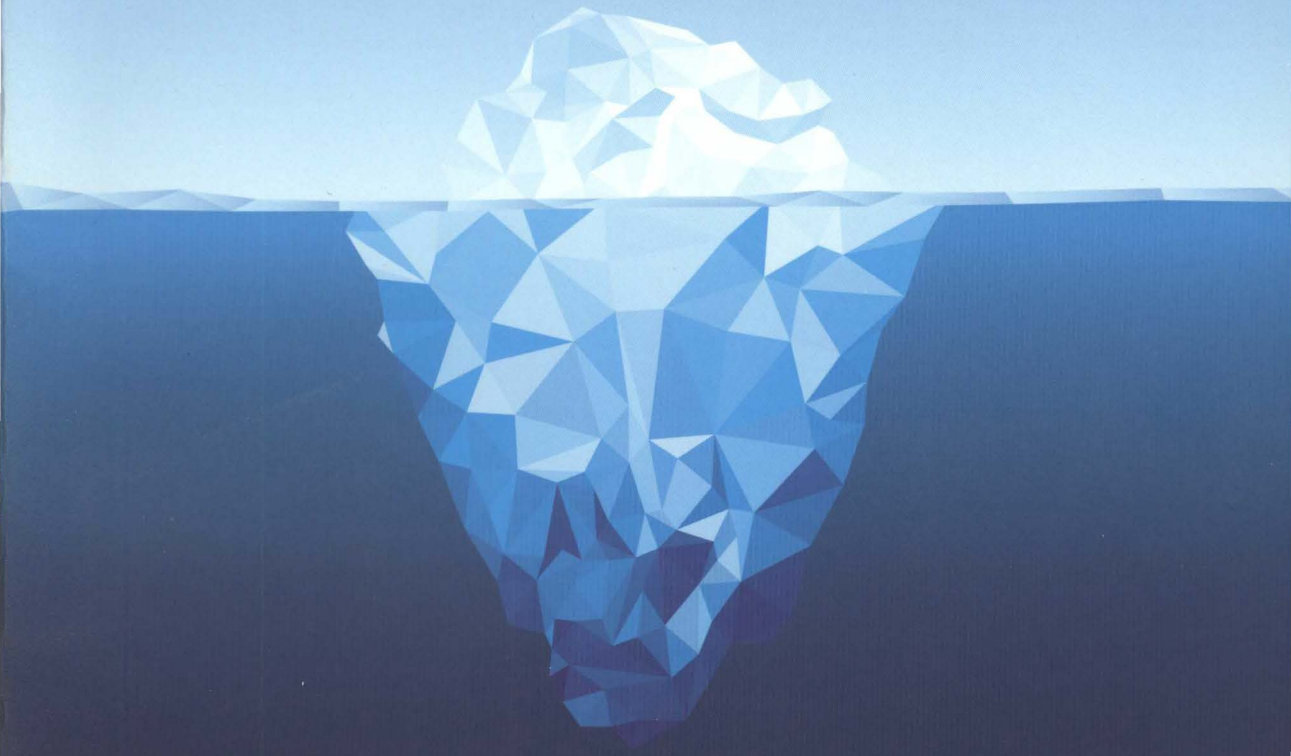
从本书中你能收获到的不仅仅只是架构理论，
更多的是来自互联网场景下大型网站架构演变过程中
核心技术难题的解决方案。

Broadview®
www.broadview.com.cn



人人都是架构师

分布式系统架构落地与瓶颈突破

高翔龙 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

高翔龙

杭州云集微店架构师，基础架构组负责人，负责基础技术平台的架构设计和中间件研发等工作，技术书籍《Java虚拟机精讲》作者，热衷于开源技术，常年游走在Github上。

人人都是架构师

分布式系统架构落地与瓶颈突破

高翔龙 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书并没有过多渲染系统架构的理论知识，而是切切实实站在开发一线角度，为各位读者诠释了大型网站在架构演变过程中出现一系列技术难题时的解决方案。本书首先从分布式服务案例开始介绍，重点为大家讲解了大规模服务化场景下企业应该如何实施服务治理；然后在大流量限流/消峰案例中，笔者为大家讲解了应该如何有效地对流量实施管制，避免大流量对系统产生较大冲击，确保核心业务的稳定运行；接着笔者为大家讲解了分布式配置管理服务；之后的几章，笔者不仅为大家讲解了秒杀、限时抢购场景下热点数据的读/写优化案例，还为大家讲解了数据库实施分库分表改造后所带来的一系列影响的解决方案。

本书适用于任何对分布式系统架构感兴趣的架构师、开发人员以及运维人员。相信阅读本书你将会有知其然和知其所以然的畅快感。

图书在版编目（CIP）数据

人人都是架构师：分布式系统架构落地与瓶颈突破 / 高翔龙著. —北京：电子工业出版社，2017.5
ISBN 978-7-121-31238-0

Ⅰ. ①人… Ⅱ. ①高… Ⅲ. ①分布式计算机系统－系统设计 Ⅳ. ①TP338.8

中国版本图书馆 CIP 数据核字（2017）第 066402 号

策划编辑：孙学瑛
责任编辑：徐津平
特约编辑：赵树刚
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：12.5　字数：220 千字
版　　次：2017 年 5 月第 1 版
印　　次：2017 年 5 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

本书的创作初衷

任何一本书，都是一个用于承载知识的载体，读者可以从中探寻自己想要知道的答案。对于我而言，书本就是带我领略奇妙计算机世界最快的一条途径。之所以想创作一本与大型分布式系统架构相关的书籍，是因为我在最近几年的实际工作中经历了太多的技术难题。每当我和我的团队尝试解决这些问题之前，时常想着能否从市面上现有的架构书籍中寻求到解决方案；但事与愿违，目前市面上高歌架构理论的读物居多，而真正讲解大型网站在架构演变过程中出现技术难题时应该如何解决的书籍却寥寥无几。对于这块领域的空白，我想尝试着去创作，尽量把我自己脑海中的内容写出来，让更多人受益，毕竟架构是需要落地的，否则便是一纸空谈。

本书内容重点

本书每一章的内容几乎都是独立的，大家完全可以挑选自己感兴趣或者有需要的部分进行阅读。本书一共包含 5 章，笔者首先从分布式服务案例开始讲起，将大家带进分布式系统的殿堂。在第 1 章中，笔者讲解了大型网站的架构演变过程，让大家对分布式系统建立一个基本的认识。当然，本章的重点是讲解企业在大规模服务化后应该如何实施服务治理，以及应该如何构建一个分布式调用跟踪系统，以一

种可视化的方式来展现跟踪到的每一个请求的完整调用链，并收集调用链上每个服务的执行耗时，整合孤立日志等。

为了避免大促场景下峰值流量过大，对系统造成较大负载导致产生雪崩现象，笔者在本书的第 2 章为大家讲解了大流量限流/消峰案例，让系统的负载压力始终处于一个比较均衡的水位，从而保护系统的稳定运行。笔者首先从限流算法开始讲起，然后分享了业务层面和技术层面等两个维度的流量管制方案。当然，本章的重点是为大家演示如何通过 MQ 来实现大流量场景下的流量消峰。

本书的第 3 章为大家讲解了分布式配置管理服务案例（配置中心）。尽管目前一些中小型互联网企业仍然将本地配置作为首选，但是当网站发展到一定规模后，继续采用本地配置所暴露的问题将会越来越多。大型网站使用分布式配置管理平台不仅能够实现配置信息的集中式管理、降低维护成本和配置出错率，还能够动态获取/更新配置信息。本章的重点是为大家演示如何基于 ZooKeeper 构建一个分布式配置管理平台，以及使用淘宝 Diamond 和百度 Disconf 系统来实现分布式配置管理服务。

热点数据的读/写操作其实是秒杀、限时抢购场景下最核心的技术难题。在大促场景下，由于峰值流量较大，大量针对同一热卖商品的并发读/写操作一定会导致后端的存储系统产生性能瓶颈，因此第 4 章为大家讲解了大促场景下热点数据的读/写优化案例。尽管商品信息可以缓存在分布式缓存中，通过集群技术，可以在理论上认为其容量是无限的，但是对于大促场景下的热卖商品来说，由于单价比平时更给力、更具吸引力，因而自然会比平时吸引更大的流量进来；这时同一个 Key 必然会落到同一个缓存节点上，而分布式缓存在这种情况下一定会出现单点瓶颈，因此笔者为大家演示了如何实施多级 Cache 方案来防止分布式缓存系统出现单点瓶颈。由于写操作无法直接在缓存中完成，因此大量的并发更新热点数据（库存扣减）都是针对数据库中同一行的——本书以 MySQL 为例，而这必然会引起大量的线程来相互竞争 InnoDB 的行锁；并发越大时，等待的线程就越多，这会严重影响数据库的 TPS，导致 RT 线性上升，最终可能引发系统出现雪崩。为了避免数据库沦为瓶颈，笔者为

大家演示了如何通过分布式锁、乐观锁在分布式缓存系统中扣减库存、通过抢购限流控制单机并发写流量，以及如何使用阿里开源的 AliSQL 数据库提升“秒杀”场景性能。

在本书的最后一章，笔者为大家讲解了数据库分库分表案例。本章演示了如何通过分库分表中间件 Shark 来帮助企业实施分库分表改造，以及分库分表后所带来一系列影响的解决方案，并重点分享了笔者在实际工作中订单业务实施分库分表改造后，应该如何同时满足 Buyer 和 Seller 的多维度查询需求。

本书面向的读者

本书适用于任何对分布式系统架构感兴趣的架构师、开发人员以及运维人员。笔者尽量用通俗易懂的文字描绘本书的各个知识点，并引用了大量在实际工作中笔者遇到的那些真实案例，相信阅读本书时你将会有知其然并知其所以然的畅快感。

读者讨论

由于笔者能力有限，书中难免会出现一些错误或者不准确的地方，你可以通过邮箱 gao_xianglong@sina.com 将问题反馈给我，我会尽量对所有问题都给予答复。

致谢

首先我要感谢我们家莹宝宝，是你的支持和鼓励才让我有了继续创作下去的勇气。还记得在本书的创作过程中，每当我写完一节时，我都会“强迫”你高声朗读帮我梳理下笔的准确度；以及每当我头痛欲裂思绪全无时，你的陪伴点燃了我在每个凌晨的斗志；甚至在我烦躁时，你总是毫无怨言地忍受着我的“坏脾气”。谢谢你的包容和体贴，我爱你。

其次我要感谢我的团队：我的两位 BOSS——冰冰和校长，最牛的 MySQL DBA 平哥，架构师大飞、青龙、小狼、僧哥、布爸，感谢你们平时在工作上的支持。

当然，本书能够顺利出版，离不开本书的两位编辑：孙学瑛老师和 Anna 老师的共同努力；感谢你们辛苦的文字校对工作，同时也祝愿孙学瑛老师家的猴宝宝健康茁壮地成长。

最后感谢那些曾经帮助过我的所有人，我爱你们！

高翔龙

2016 年 12 月 31 日深夜

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **提交勘误**：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与我们交流**：在页面下方【读者评论】处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31238

二维码：

目录

1 第1章 分布式服务案例

写一本书，远远不是大家想象得那么简单，更不是思绪如泉涌般的信手拈来，其中的艰辛和酸甜苦辣想必只有作者自己才能够深刻体会到。笔者在创作本书时，可谓是下笔艰难、字斟句酌，当然这一切都是为了能够将笔者想说的和想展现的内容毫无保留地诠释给各位读者。本章作为本书的开篇，也是笔者深思熟虑后的结果。目前，在互联网场景下应对大流量、高并发，服务化架构改造是必不可少的。在本章中，笔者为大家讲解了互联网场景下大型网站架构的演变过程、服务化的一些必备基础知识，以及如何使用开源社区流行的 RPC 和服务治理框架 Dubbo 帮助企业落地服务化架构。当然，本章的重点是为大家演示如何实现一个拥有较低侵入性的分布式调用跟踪系统来帮助企业实施服务治理，因为在一些大规模的服务调用场景下，我们必然需要一种有效且可视化的手段来帮助开发人员、架构师更好地梳理清楚服务或服务之间的依赖关系、调用顺序，以及服务的执行耗时等。

OK，接下来就请大家跟随笔者一起来探询分布式场景下服务化的真谛吧！Let's GO!

1.1 分布式系统的架构演变过程

在移动互联网的浪潮中，你我正生逢其时地享受着当下，如果你愿意做一只站在风口上等待起飞的猪，那么请认真地问问自己，是否已经准备好了？互联网究竟是什么？简而言之，互联网诠释的是一种精神，融入了高度开放、分享，以及自由的精神。如果你想融入这个圈子，那么请务必先舍弃掉与互联网精神背道而驰的陈旧观念和思维，否则你自始至终都只会被孤立在局外。

互联网悄然改变了世界，改变了人们对事务的认知，缩短了人与人之间的距离。无论你是否愿意承认，互联网已经完全影响并融入我们的生活中。我们的长辈们，也从早期对新鲜事物的排斥，变成现在的欣然接受，这就是互联网与生俱来的魅力和魔力。笔者的母亲从来就不是一个喜欢追赶潮流的人，但是她早已智能设备从不离身，每天早上起床的第一件事情就是拿起智能手机，刷刷朋友圈、看看时事政治、做回“吃瓜群众”，八卦下娱乐新闻，甚至衣食住行也几乎是通过互联网这个载体一键搞定的，如图 1-1 所示。既然互联网能使我们的生活质量更好，那就请张开双臂紧紧拥抱它。

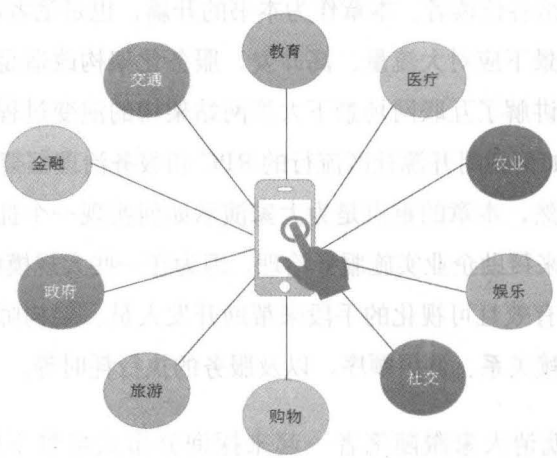


图 1-1 拥有互联网的生活

拥有互联网的夜晚是明亮的，当然为这寂静夜空点燃光明的正是聚集在各个互联网企业内部的技术团队，正是这些家伙昼夜颠倒的辛勤付出（无数的通宵、无数的会议、无数的版本迭代、无数的交互体验优化、无数的系统架构升级），才换来今日互联网的光彩夺目。不过任何事物都如同硬币一般具备两面性，我们不能“光见贼吃肉，不见贼挨打”，这句话放在互联网领域似乎相当应景，很多电商网站为了吸引用户流量，往往都会以极低的价格作为诱饵，但是当促销活动正式开始后，峰值流量却远远超出了系统所能够承受的处理能力，这必然只会产生一种结果——宕机。如何帮助企业顺利走出困境，打造真正具备高性能、高可用、易扩展、可伸缩，以及相对安全的网站架构才是架构师、技术大牛们应该重点思考的问题和责任，哪怕是系统宕机，也要尽最大努力做到“死而不僵”，用技术支撑业务的野蛮生长，活下去才会有更美好的明天。

一般来说，网站由小变大的过程，几乎都需要经历单机架构、集群架构、分布式架构。伴随着业务系统架构一同演变的还有各种外围系统和存储系统，比如关系型数据库的分库分表改造、从本地缓存过渡到分布式缓存等。当系统架构演变到一定阶段且逐渐趋向于稳定和成熟后，架构师们需要对技术细节追本溯源，如果现有的技术或者框架不能有效满足业务需要，就需要从“拿来主义”的消费者角色转变为自行研发的生产者角色。当然，在这条技术之路离你和你所在的企业还很遥远的时候，尽管未雨绸缪利大于弊，但这却并不是你现阶段的工作重点。在此大家需要注意，对技术由衷的热爱和痴迷本身并没有什么不对，但是千万不能够过于沉醉而选择刻意忽略掉业务，任何技术的初衷都是为了更好地服务业务，一旦脱离业务，技术必然会失去原有的价值和意义，切勿舍本逐末。

1.1.1 单机系统

任何一个网站在发布初期几乎都不可能立马就拥有庞大的用户流量和海量数据，都是在不停的试错过程中一步一步演变其自身架构，满足其自身业务。所以我

们常说，互联网领域几乎没有哪一个网站天生就是大型网站，因为往往系统做大与业务做大是呈正比的，大型网站都是从小型网站逐渐演变过来的，而不是被刻意设计出来的。试想一下，如果业务不见起色，一味地追求大型网站架构又有何意义呢？

对于一个刚上线的项目，我们往往会将 Web 服务器、文件服务器和数据库全都部署在同一台物理服务器上，这样做的好处只有一个——省钱，如图 1-2 所示。资金紧张且用户流量相对较小的网站，采用这样的架构进行部署确实非常实惠，毕竟用户流量上不去，自然就没有必要考虑更多的问题，哪怕是系统宕机，影响范围也不大。当然，一旦业务开始加速发展，用户逐渐增多，系统瓶颈便会开始暴露，这时架构师就可以考虑对现有网站架构做出以下四点调整：

- 独立部署，避免不同的系统之间相互争夺共享资源（比如 CPU、内存、磁盘等）；
- Web 服务器集群，实现可伸缩性；
- 部署分布式缓存系统，使查询操作尽可能在缓存命中；
- 数据库实施读/写分离，实现 HA（High Availability，高可用性）架构。

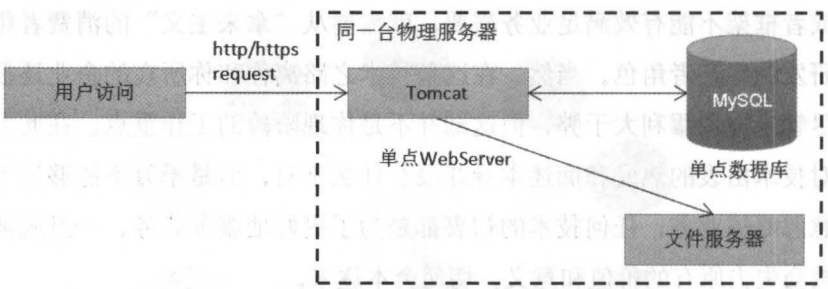


图 1-2 在同一台机器上部署单机系统

1.1.2 集群架构

一旦用户开始增多，并发流量上来后，为了让用户拥有更好的操作体验，我们

不得不对单机系统架构做出调整和优化，因此在这个阶段主要需要解决的问题就是提升业务系统的并行处理能力，降低单机系统负载，以便支撑更多的用户访问操作。

集群（Cluster）技术可以将多台独立的服务器通过网络相互连接组合起来，形成一个有效的整体对外提供服务，使用集群的意义就在于其目标收益远高于所付出的实际成本和代价。一般，互联网领域都有一个共同的认知，那就是当一台服务器的处理能力接近或已超出其容量上限时，不要企图更换一台性能更强劲的服务器，通常的做法是采用集群技术，通过增加新的服务器来分散并发访问流量，1 台不够就扩到 2 台，2 台不够就扩到 4 台，只要业务系统能够随意支持服务器的横向扩容，那么从理论上来说就应该无惧任何挑战，从而实现可伸缩性和高可用性架构。

如图 1-3 所示，对于无状态的 Web 节点来说，通过 Nginx 来实现负载均衡调度似乎是一个不错的选择，但是在生产环境中，Nginx 也应该具备高可用性，这可以依靠 DNS 轮询来实现。在集群环境中，Web 节点的数量越多，并行处理能力就越强，哪怕其中某些节点因为种种原因宕机，也不会使系统的整体服务不可用。

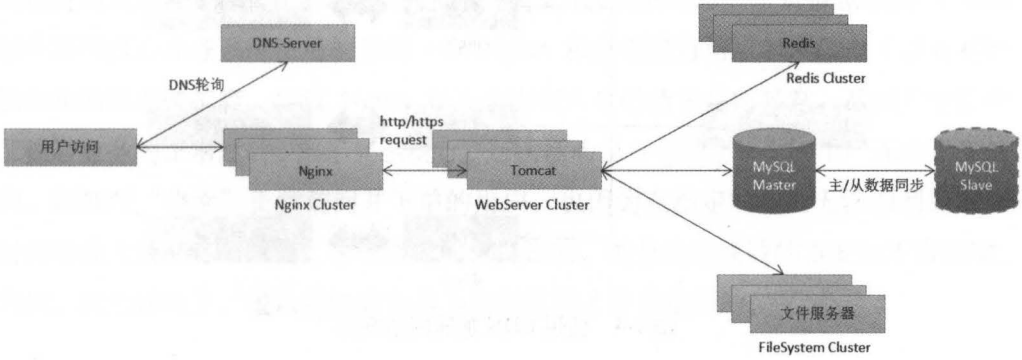


图 1-3 在独立的机器上部署集群系统

伴随着 Web 集群改造的还有分布式缓存和数据库，对于查询操作我们应该尽可能在缓存命中，从而降低数据库的负载压力。尽管缓存技术可以解决数据库的大部

分查询压力，但是写入操作和无法在缓存命中的数据仍然需要频繁地对数据库进行读/写操作，因此对数据库实施读/写分离改造也迫在眉睫。

业务发展到一定阶段后必然会变得更加复杂，用户规模也会线性上升，这时架构师就可以考虑对现有网站架构做出以下两点调整：

- 利用 CDN 加速系统响应；
- 业务垂直化，降低耦合，从而实现分而治之的管理。

由于中国的网络环境相对比较复杂，跨网络、跨地域的用户访问网站时，速度有较大差别，因此当用户流量增大之后，我们需要考虑将系统中的一些静态资源数据（如图片、音频、视频、脚本文件及 HTML 网页等）缓存在 CDN 节点上，因为 CDN 正变得越来越廉价，如图 1-4 所示。由于用户的请求并不会直接落到企业的数据中心，而是请求到离用户最近的 ISP（Internet Service Provider，互联网服务提供商）上，因此可以大幅提升系统整体的响应速度。

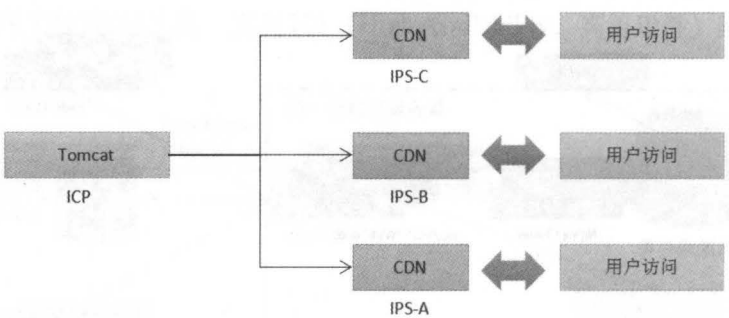


图 1-4 使用 CDN 加速网站响应

1.1.3 拆系统之业务垂直化

尽管业务系统可以通过集群技术来提升并行处理能力和实现高可用架构，但是一般到了这个阶段，不仅用户规模暴增，业务需求也变得更加复杂。由于业务逻辑

全部耦合在一起，并且部署在同一个 Web 容器中，这必然会导致系统中不同业务之间的耦合过于紧密，除了扩展和维护困难，在生产环境中极有可能会因为某个业务功能不可用而影响系统整体服务不可用，因此在这个阶段主要需要解决的问题就是降低业务耦合，实现高内聚低耦合，提升系统容错性，避免牵一发而动全身的风险。

下面为大家分享笔者亲身经历过的一次真实生产事故。

在一次“双 11”抢购活动中，我们预计零点开抢时的那一拨峰值流量肯定会远高于平时，因此在几天前运维部门的同学就提前扩容了上百台服务器来应对“双 11”。在活动开始前，大家摩拳擦掌，屏住呼吸紧盯着电脑屏幕“观赏”流量监控曲线图时，悲催的事情发生了。随着倒数结束，活动正式开始，流量曲线居然没有太大的起伏，这让当时在场的小伙伴们瞬间懵掉了，紧急排查故障后发现，前端 APP 在每一次业务请求提交之前都会先将客户端埋点数据上报到专门用于数据收集的 Web 服务器上（后期需要利用大数据平台对埋点数据进行一些用户行为的实时/离线分析），然后再请求到业务系统上执行逻辑处理。但因为我们的疏忽，负责数据收集的 Web 服务器和核心业务系统使用的是同一个 Nginx 服务器进行请求转发，由于没有对数据收集的机器做扩容，导致 Nginx 日志中出现大量的请求超时异常，从而严重影响了核心业务的正常运转。据不完全统计，这次活动至少导致了 2/3 的用户无法正常访问，而那些“侥幸”正常访问并下单的用户，也因为在指定时间内无法顺利完成支付而导致大量的订单回流。这次惨痛的教训说明，细节决定成败的重要性不言而喻。当然，既然踩坑了，吸取经验教训避免悲剧重现才是最重要的。

大家一定要具备大系统小做的意识，所谓拆系统其实指的就是业务垂直化，简而言之，架构师可以根据系统业务功能的不同拆分出多个业务模块（一般大型电商网站都会拆分出首页、用户、搜索、广告、购物、订单、商品、收益结算等子系统），再由不同的业务团队负责承建，分而治之，独立部署，如图 1-5 所示。在此大家需要

注意，拆分粒度越细，耦合越小，容错性越好，每个业务子系统的职责就越清晰。但是如果拆分粒度过细，维护成本将是一个不小的挑战。对业务系统完成拆分后，就意味着系统已经过渡到分布式系统了，这也为企业实施服务化改造和数据库分库分表改造提前做好了准备。关于数据库分库分表的相关知识点，大家可以直接阅读本书的第 5 章。

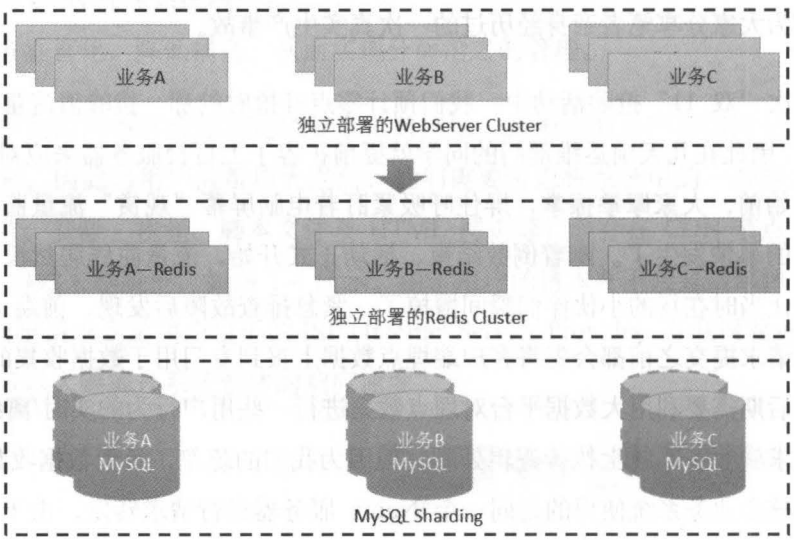


图 1-5 业务垂直化改造

1.1.4 为什么需要实现服务化架构

在大部分互联网企业的系统架构演变过程中，不得不提到的就是服务化改造，那么究竟什么是服务化，以及为什么要实现服务化架构呢？

随着用户规模逐渐庞大，需求更加复杂，我们一定会对耦合在一个 Web 容器中的业务系统进行垂直化改造，以业务功能为维度拆分出多个子系统，这样做就是为了能够更清晰地规划和体现出每个子系统的职责，降低业务耦合，以及提升容错性。

但是在多元化的业务需求下，子系统中一定会存在较多的共享业务，这些共享业务肯定会被重复建设，产生较多的冗余业务代码。而且，业务系统中数据库连接之类的底层资源必然会限制业务系统所允许横扩的节点数量，因为在横扩过程中，连接数是机器数的平方（若机器数为 1，则连接池中的连接数为 2；机器数为 2，则连接池中的连接数为 4；机器数为 3，则连接池中的连接数为 9，以此类推）。除了共享业务重复建设和资源连接受限，还有一个不容忽视的问题，当业务做大时，技术团队的人员规模也开始膨胀，太多人共同维护一个系统肯定会坏事，尤其是那些“手潮”的同学，经常会导致 SVN 中的版本冲突。因此为了避免这些问题，服务化架构（Service Oriented Ambiguity，SOA）改造刻不容缓，如图 1-6 所示。

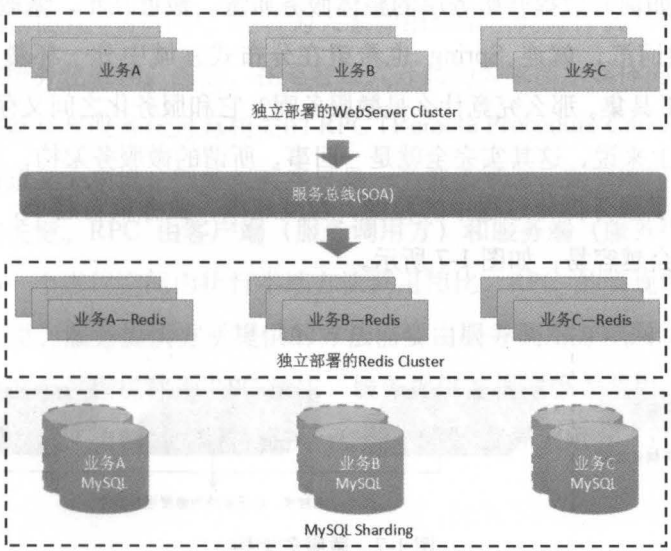


图 1-6 服务化架构

当然，如果你现阶段并不满足上述任何一个条件，笔者其实是不建议实施服务化改造的，因为这显然会提升系统的复杂度和维护成本，甚至还需要提前规划一些服务化场景下才需要考虑的技术难题，如分布式事务、服务治理等。

1.1.5 服务拆分粒度之微服务

业务系统实施服务化改造后，原本共享的业务被拆分，形成可复用的服务，可以在最大程度上避免共享业务的重复建设、资源连接瓶颈等问题出现。那么那些被拆分出来的服务，是否也需要以业务功能为维度来进行拆分，使之能够独立进行部署，以降低业务耦合和提升容错性呢？

由于 Web 容器中子系统的服务层被剥离，因此这些子系统从某种意义上来说仅仅只是一个控制层（Controller），由于控制层无须处理任何复杂的业务逻辑，因此吞吐量自然会得到提升，但是业务的逻辑处理由独立部署的服务层负责，所以架构师需要认真思考如何有效提升服务层的整体服务质量。最近几年，微服务绝对是一个被炒得火热的词汇，就连 Spring 也希望在分布式领域中分一杯羹，借此推出了 Spring-Cloud 工具集。那么究竟什么是微服务呢？它和服务化之间又存在什么必然联系呢？从本质上来说，这其实完全就是一回事，所谓的微服务架构，从宏观上来看，无非就是细化了服务拆分过程中的粒度，粒度越细，业务耦合越小，容错性越好，并且后期扩展会越容易，如图 1-7 所示。

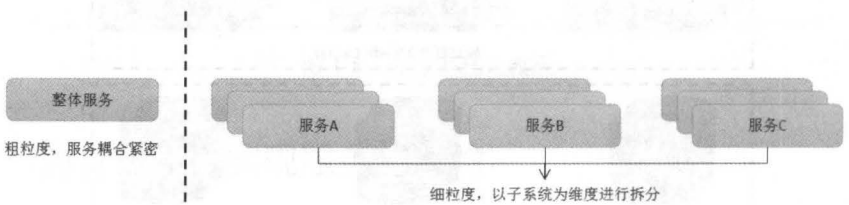


图 1-7 微服务架构

那么服务到底应该如何拆分才称得上“微服务”呢？一般来说，笔者建议大家以子系统为维度进行服务拆分，也可以在此基础上对服务的读/写进行分离，或者拆分出单独的服务，这样服务的拆分粒度大小适中，所付出的成本与实际收益会相对更加均衡，否则粒度过细不仅会导致维护成本提升，而且系统出现问题时定位和排查问题困难重重，更重要的是很难梳理服务之间的依赖关系。

1.2 系统服务化需求

既然服务化能够解决在拆分子系统时遇到的一些问题和瓶颈，那么接下来笔者就为大家详细讲解服务化的具体实施细节。笔者先为大家讲解服务化与 RPC 协议的具体关系，再实战演示目前在开源社区服务化领域最为开发人员所熟知的阿里分布式服务框架 Dubbo，以及企业在实施服务化改造后对现有系统产生的一些影响的解决方案。

1.2.1 服务化与 RPC 协议

在本章的前面几个小节中，笔者为大家详细介绍了究竟什么是服务化，以及为什么需要落地服务化架构。当然，在笔者正式开始为大家演示具体的实施细节之前，不得不提及的就是与服务化息息相关的 RPC（Remote Procedure Call，远程过程调用）协议。从严格意义上来说，服务化其实只是一个抽象概念，而 RPC 协议才是用于实现服务调用的关键。RPC 由客户端（服务调用方）和服务端（服务提供方）两部分构成，和在同一个进程空间内执行本地方法调用相比，RPC 的实现细节会相对复杂不少。简单来说，服务提供方所提供的方法需要由服务调用方以网络的形式进行远程调用，因此这个过程也称为 RPC 请求，服务提供方根据服务调用方提供的参数执行指定的服务方法，执行完成后再将执行结果响应给服务调用方，这样一次 RPC 调用就完成了。

服务化框架的核心就是 RPC，目前市面上成熟的 RPC 实现方案有很多，比如 Java RMI、WebService、Hessian 及 Finagle 等。在此需要注意，不同的 RPC 实现对序列化和反序列化的处理也不尽相同，比如将对象序列化成 XML/JSON 等文本格式尽管具备良好的可读性、扩展性和通用性，但却过于笨重，不仅报文体积大，解析过程也异常缓慢，因此在一些特别注重性能的场景下，采用二进制协议更合适。看到这

里或许有些人已经产生了疑问，实现服务调用无非就是跨进程通信而已，那是否可以使用 Socket 技术自行实现呢？带着疑问，笔者来为大家梳理一下完成一次 RPC 调用主要需要经历的三个步骤：

- 底层的网络通信协议处理；
- 解决寻址问题；
- 请求/响应过程中参数的序列化和反序列化工作。

其实，RPC 的本质就是屏蔽上述复杂的底层处理细节，让服务提供方和服务调用方都能够以一种极其简单的方式（甚至简单到就像是在实现一个本地方法和调用一个本地方法一样）来实现服务的发布和调用，使开发人员只需要关注自身的业务逻辑即可，如图 1-8 所示。

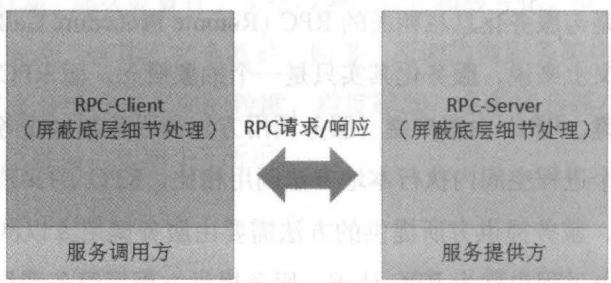


图 1-8 RPC 请求的调用过程

1.2.2 使用阿里分布式服务框架 Dubbo 实现服务化

由于 RPC 协议屏蔽了底层复杂的细节处理，并且随着后续服务规模的扩大要考虑服务治理等众多因素，因此笔者在生产环境中落地服务化架构时所使用的 RPC 框架就是阿里开源的分布式服务框架 Dubbo。毫不客气地说，Dubbo 真的是集万千宠爱于一身，目前大部分互联网企业都偏爱使用 Dubbo 来落地服务化架构，就是因为 Dubbo 的设计精良、使用简单、技术文档丰富。而且，Dubbo 预留了足够多的接口使

开发人员能够非常容易地对 Dubbo 进行二次开发，以便更好地满足业务需求，所以 Dubbo 在开源社区拥有众多的技术拥护者和推进者。但是由于一些特殊原因，目前 Dubbo 的主干代码已经停止了更新（现在阿里内部主推 HSF），当然这并不意味着我们在使用上得不到保障（目前 Dubbo 的版本已经相当稳定，而且由于很多坑别人都已经踩过，大家在使用过程中根本不需要去填坑，参考开源社区的解决方案即可），当当网基于 Dubbo 二次开发的 Dubbox 框架在开源社区同样广受好评。

如图 1-9 所示，Provider 作为服务提供方负责对外提供服务，当 JVM 启动时 Provider 会被自动加载和启动，由于 Provider 并不需要依赖任何的 Web 容器，因此它可以运行在任何一个普通的 Java 程序中。当 Provider 成功启动后，会向注册中心注册指定的服务，这样作为服务调用方的 Consumer 在启动后便可以向注册中心订阅目标服务（服务提供者的地址列表），然后在本地根据负载均衡算法从地址列表中选择其中的某一个服务节点进行 RPC 调用，如果调用失败则自动 Failover 到其他服务节点上。

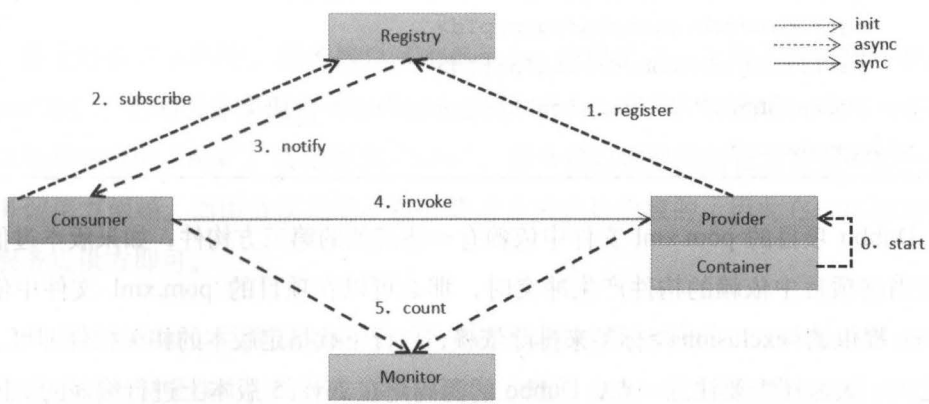


图 1-9 Dubbo 服务调用

在此需要注意，在服务的调用过程中还存在一个不容忽视的问题，即监控系统。试想一下，如果在生产环境中业务系统和外围系统没有部署相对应的监控系统来帮

助开发人员分析和定位问题，那么这与脱了缰的野马无异，当问题出现时必然会导致开发人员手足无措，相互之间推卸责任，因此监控系统的重要性不言而喻。值得庆幸的是，Dubbo 为开发人员提供了一套完善的监控中心，使我们能够非常清楚地知道指定服务的状态信息（如服务调用成功次数、服务调用失败次数、平均响应时间等），如图 1-10 所示。这些数据由 Provider 和 Consumer 负责统计，再实时提交到监控中心。

Statistics (1)					
Method:	Success	Failure	Avg Elapsed (ms)	Max Elapsed (ms)	Max Concurrent
orderQuery	507 --> 0	0 --> 0	109 --> 0	4399 --> 0	35 --> 0

图 1-10 Dubbo 服务状态监控

接下来笔者就为大家演示 Dubbo 的基本使用，本书使用的 Dubbo 版本为 2.5.3，开发人员可以通过 Maven 依赖的方式下载 Dubbo 构件，如下所示：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>dubbo</artifactId>
    <version>2.5.3</version>
</dependency>
```

Dubbo 项目的 pom.xml 文件中依赖有一些其他的第三方构件，如果版本过低或者与当前项目中依赖的构件产生冲突时，那么可以在项目的 pom.xml 文件中使用 Maven 提供的<exclusions/>标签来排除依赖，自行下载指定版本的相关构件即可。除此之外，大家还需要注意一点，Dubbo 的源码是在 Java 5 版本上进行编译的，因此在项目中使用 Dubbo 时，JDK 版本应该高于或等于 Java 5。

成功下载好运行 Dubbo 所需的相关构件后，首先要做的事情就是定义服务接口和服务实现，如下所示：

```
package com.xx.xx;
public interface UserService {
    public boolean login(String account, String pwd);
}
package com.xx.xx.provider;
public class UserServiceImpl implements UserService {
    @Override
    public boolean login(String account, String pwd) {
        boolean result = false;
        if ("admin".equals(account)) {
            if ("123456".equals(passWord)) {
                result = true;
            }
        }
        return result;
    }
}
```

在上述程序示例中，服务接口 UserService 中提供了一个用于模拟用户登录的login()方法，它的服务实现为 UserServiceImpl 类。如果用户正确输入账号和密码，那么结果将返回“true”，否则返回“false”。服务接口需要同时包含在服务提供方和服务调用方两端，而服务实现对于服务调用方来说是隐藏的，因此它仅仅需要包含在服务提供方即可。

从理论上来说，尽管 Dubbo 可以不依赖任何的第三方构件，只需有 JDK 的支撑就可以运行，但是在实际的开发过程中，为了避免 Dubbo 对业务代码造成侵入，笔者推荐大家将 Dubbo 集成到 Spring 中来实现远程服务的发布和调用。在 Provider 的 Spring 配置信息中发布服务，如下所示：

```
<dubbo:application name="provider-test" />
```

```
<!-- 配置注册中心地址，注册服务提供者的地址列表 -->
<dubbo:registry protocol="zookeeper" address="127.0.0.1:2181" />
<!-- 声明服务端口 -->
<dubbo:protocol name="dubbo" port="20880" />
<!-- 声明服务接口 -->
<dubbo:service interface="com.xx.xx.UserService" ref="userService" />
<bean id="userService" class="com.xx.xx.provider.UserServiceImpl" />
```

成功启动 Provider 后，便可以在注册中心看见由 Provider 发布的远程服务。在 Consumer 的 Spring 配置信息中引用远程服务，如下所示：

```
<dubbo:application name="consumer-test" />
<!-- 配置注册中心地址，获取服务提供者的地址列表 -->
<dubbo:registry protocol="zookeeper" address="127.0.0.1:2181" />
<!-- 声明远程服务代理 -->
<dubbo:reference id="userService" interface="com.xx.xx.UserService" />
```

成功启动 Consumer 后，便可以对目标远程服务进行 RPC 调用（使用 Dubbo 进行服务的发布和调用，就像实现一个本地方法和调用一个本地方法一样简单，因此笔者省略了服务调用的相关代码）。关于 Dubbo 的更多使用方式，本书不再一一进行讲解，大家可以参考 Dubbo 的用户指南：http://dubbo.io/User+Guide-zh.htm。

1.2.3 警惕 Dubbo 因超时和重试引起的系统雪崩

在上一个小节中，笔者为大家演示了 Dubbo 框架的一些基本使用方法，尽管使用 Dubbo 来实现 RPC 调用非常简单，但是刚接触 Dubbo 框架的开发人员极有可能因为对其不熟悉而跌进一些“陷阱”中。因此，开发人员需要重视看似微不足道的细节，这往往可以非常有效地避免系统在大流量场景下产生雪崩现象。在对数据库、分布式缓存进行读/写访问操作时，我们往往都会在程序中设置超时时间，一般笔

者不建议在生产环境中将超时时间设置得太长，否则如果应用获取不到会话，又长时间不肯返回，那么一定会对业务产生较大的影响。但将超时时间设置得太短又可能适得其反，因此超时时间究竟应该如何设置需要根据实际的业务场景而定，大家可以在日常的压测过程中仔细评估。

为 Dubbo 设置超时时间应该是有针对性的，比较简单的业务执行时间较短，可以将超时时间设置得短一点，但对于复杂业务而言，则需要将超时时间适当地设置得长一点。笔者之前曾经提及过，Dubbo 的 Consumer 会在本地根据负载均衡算法从地址列表中选择某一个服务节点进行 RPC 调用，如果调用失败则自动 Failover 到其他服务节点上，默认重试次数为两次，服务调用超时就意味着调用失败，需要进行重试，如图 1-11 所示。在此需要注意，如果一些复杂业务本身就需要耗费较长的时间来执行，但超时时间却被不合理地设置为小于服务执行的实际时间，那么在大流量场景下，系统的负载压力将被逐步放大，最终产生蝴蝶效应。假设有 1000 个并发请求同时对服务 A 进行 RPC 调用，但都因超时导致服务调用失败，由于 Dubbo 默认的 Failover 机制，共将产生 3000 次并发请求对服务 A 进行调用，这是系统正常压力的 3 倍，若处于峰值流量时情况可能还会更糟糕，大量的并发重试请求很可能直接将 Dubbo 的容量撑爆，甚至影响到后端存储系统，导致资源连接被耗尽，从而引发系统出现雪崩。

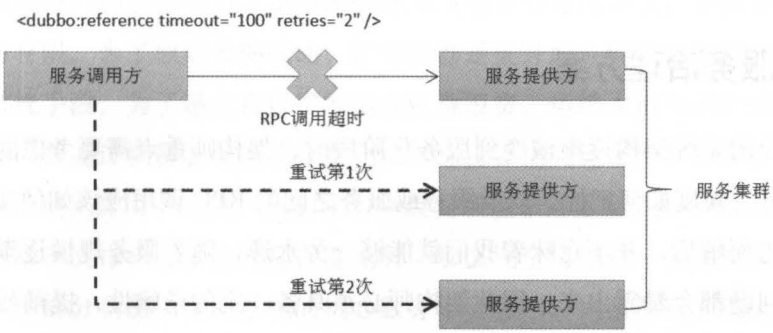


图 1-11 Dubbo 的 Failover 机制

还有一点很重要，并不是任何类型的服务都适合 Failover 的，比如写服务，由于需要考虑幂等性，因此笔者建议调用失败后不应该进行重试，否则将导致数据被重复写入。只有读服务开启 Failover 才会显得有意义，既然不需要考虑幂等性，就可以通过 Failover 来提升服务质量。

除了 Failover，Dubbo 也提供了其他容错方案供开发人员参考，如下所示：

```
<!-- 服务调用失败时，重试其他服务节点，通常用于读操作 -->
<dubbo:registry cluster="failover" />
<!-- 只发起一次调用，失败立即报错，通常用于非幂等性的写操作 -->
<dubbo:registry cluster="failfast" />
<!-- 失败安全，出现异常时直接忽略，通常用于写入审计日志等操作 -->
<dubbo:registry cluster="failsafe" />
<!-- 失败自动恢复，后台记录失败请求，定时重发，通常用于消息通知操作 -->
<dubbo:registry cluster="failback" />
<!-- 并行调用多个服务节点，成功 1 个即返回，通常用于实时性要求较高的读操作 -->
<dubbo:registry cluster="forking" />
```

上述相关参数除了可以在服务调用方配置，也适用于服务提供方，如果服务调用方和服务提供方配置有相同的参数，默认以服务调用方的配置信息为主。关于更多配置信息，本书就不再一一进行讲解，大家可以参考 Dubbo 的配置参考手册。

1.2.4 服务治理方案

当企业的系统架构逐渐演变到服务化阶段时，架构师重点需要考虑的问题是服务如何拆分、粒度如何把控，以及服务或服务之间的 RPC 调用应该如何实现。当这些问题迎刃而解后，并不意味着我们就能够一劳永逸，随着服务规模逐渐扩大，一些棘手的问题都会暴露出来，因此架构师必须具备一定的前瞻性，提前规划和准备充足的预案去应对将来可能发生的种种变故，否则这就等于给自己挖坑，与其花大

把时间去填坑，不如用更多的精力去思考企业在大规模服务化前应该如何实施服务治理。

从严格意义上来说，Dubbo 不仅是一个 RPC 框架，更是一个服务治理框架，因为 Dubbo 几乎提供了一套完整的服务治理方案，所以它从诞生起就备受瞩目和爱戴。由于服务治理涉及的范围非常广泛，可能会导致一些人对服务治理的概念比较模糊，或者根本不理解服务治理的重要性和必要性，笔者归纳的关于服务治理的三个基础要素如下所示：

- 服务的动态注册与发现；
- 服务的扩容评估；
- 服务的升/降级处理。

在 1.2.2 节中，笔者曾为大家介绍过 Dubbo 的注册中心，那么大家思考一下，为什么需要引入注册中心呢？当服务变得越来越多时，如果把服务的调用地址（URL）配置在服务调用方，那么 URL 的配置管理将变得非常麻烦，因此引入注册中心的目的就是实现服务的动态注册和发现，让服务的位置更加透明，这样服务调用方将得到解脱，并且在客户端实现负载均衡和 Failover 将会大大降低对硬件负载均衡器的依赖，从而减少企业的支出成本。部署监控中心是为了更好地掌握服务的状态信息，因为有了这些统计数据后我们才能够准确地知道指定服务的热度，毕竟单台服务器的处理能力有限。为了应对大促活动，扩容机器是提升服务器并行处理能力一个非常重要的常规手段，为了避免盲目扩容造成资源浪费，运维人员可以将统计数据作为参考指标，并通过调整服务器的权重比例来综合评估究竟需要扩容多少节点才能够合理且有效地支撑用户流量。服务的升/降级处理也非常重要，在一些特殊场景下，如果系统容量在支撑核心业务时都捉襟见肘，那么完全可以对一些次要服务进行降级处理，牺牲部分功能来保证系统的核心服务不受影响。当某些新上线的服务可能在实现上存在缺陷时，也可以采用服务降级的手段来确保系统的整体稳定性，以后

再将这些降级服务进行升级处理即可。

关于服务黑白名单、服务权限控制、服务负责人，以及服务资源调度等其他的服务治理问题，大家可以参考 Dubbo 的用户指南。而关于服务调用跟踪的问题，大家可以直接阅读 1.3 节。

Dubbo 为开发人员提供的监控中心和管理控制台都需要单独安装和部署，大家可以参考 Dubbo 的管理员指南：http://dubbo.io/Administrator+Guide-zh.htm。如图 1-12 所示，在管理控制台中服务治理包含的功能有：路由规则、动态配置、服务降级、访问控制、权重调节及负载均衡等，成功安装好 dubbo-admin 后，便可以对其进行访问和实施服务治理。

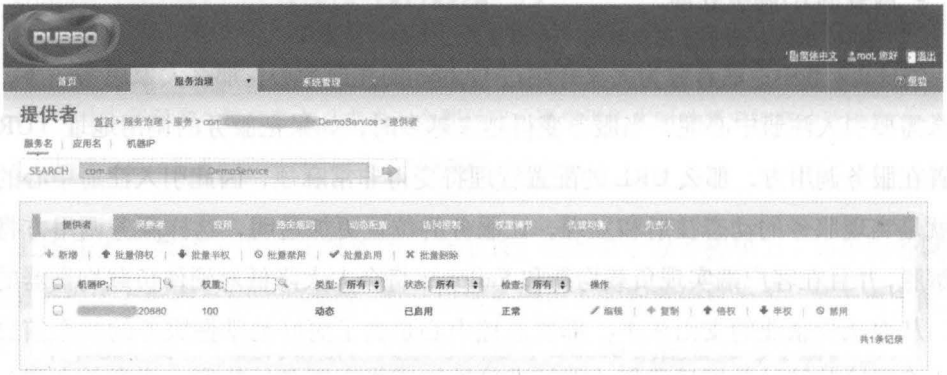


图 1-12 Dubbo 管理控制台界面

1.2.5 关于服务化后的分布式事务问题

从单机系统演变到分布式系统并不像书本中描述得那样简单，不同的业务之间必然会存在较大的差异，因此企业在实施服务化改造时肯定会困难重重，即使最终服务成功被拆分出来，架构师还需要提前思考和规划后续的服务治理等问题。当然这一切都还不是终点，就像本书封面插图所示的那座冰山，我们能够看见的问题其

实只是冰山一角，大型网站架构演变过程中等待我们解决的技术难题还有很多。大家思考一下，实施服务化改造后事务的问题应该如何解决？或许很多同学都会毫不犹豫地指出，分布式事务简直让人感到“痛心疾首”。的确，就算是银行业务系统也并不一定都采用强一致性，那么我们是否还有必要去追求强一致性呢？

其实分布式事务一直就是业界没有彻底解决的一个技术难题，没有通用的解决方案，没有高效的实现手段，但是这并不能成为我们不去解决的借口。网络上有一句非常著名的段子，“此处不留爷，自有留爷处；处处不留爷，爷走出国路”，既然分布式事务实施起来非常困难，那么我们为什么不换个思路，使用其他更优秀的替代方案呢？只要能够保证最终一致性，哪怕数据会出现不一致的短暂窗口期又有什么关系？在架构的演变过程中，哪个是主要矛盾就优先解决哪一个，就像我们对 JVM 进行性能调优一样，吞吐量和低延迟这两个目标本身就是相互矛盾的，如果吞吐量优先，那么 GC 就必然需要花费更长的暂停时间来执行内存回收；反之，频繁地执行内存回收，又会导致程序吞吐量的下降，因此大家要学会权衡和折中。关于最终一致性的实现方案，大家可以直接阅读 5.2.8 节和 5.4.2 节。

1.3 分布式调用跟踪系统需求

如图 1-13 所示，在大规模服务调用场景下，服务之间的依赖关系错综复杂，甚至连架构师都无法在短时间内梳理清楚服务之间具体的依赖关系和调用顺序。除此之外，在业务系统未实施服务化之前，各个子系统内部一定会存在较多的共享业务，尽管不利于维护，但是当某一个业务子系统出现故障时，开发人员只需要登录到指定的机器上即可快速根据错误异常日志定位问题和解决问题。随着这些共享业务被拆分成独立的服务，一次用户请求可能涉及后端多个服务之间的调用，那些分散在各个服务器上的孤立日志对于排查问题显然是非常不利的，因此到了这个阶段，企业构建分布式调用跟踪系统已迫在眉睫。

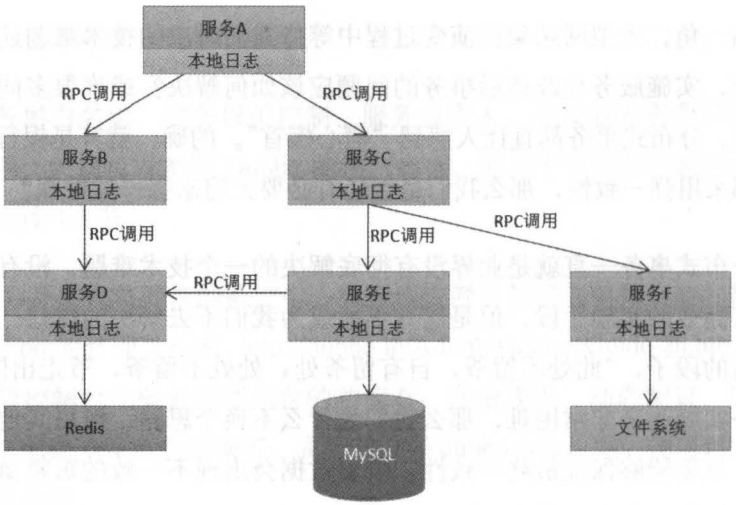


图 1-13 错综复杂的服务依赖关系

分布式调用跟踪系统其实就是一个监控平台，能够以可视化的方式展现跟踪到的每一个请求的完整调用链，以及收集调用链上每个服务的执行耗时、整合孤立日志等。目前，一些大型的互联网企业内部都构建有适用于自身业务特点的分布式调用跟踪系统，比如淘宝的鹰眼（EagleEye）、Twitter 的 Zipkin、新浪的 Watchman，以及京东的 Hydra 等。本节笔者先为大家介绍 Google 的 Dapper 论文中涉及的关于分布式调用跟踪系统的一些关键设计目标，再重点演示如何在 Dubbo 的基础上构建具备低侵入性的分布式调用跟踪系统。

1.3.1 Google 的 Dapper 论文简介

目前众所周知的一些分布式调用跟踪系统大都脱胎于 Google 的论文 *Dapper, A Large Scale Distributed Systems Tracing Infrastructure*（笔者认为翻译较好的一篇译文：http://bigbully.github.io/Dapper-translation）。这篇论文中存在一些非常有价值的参考信息，如果大家希望在企业内部构建一个适用于自身业务场景的分布式调用跟踪系统，那么笔者强烈建议仔细阅读这篇论文。

笔者从这篇论文中整理出了分布式调用跟踪系统的四个关键设计目标，如下所示：

- 服务性能低损耗；
- 业务代码低侵入；
- 监控界面可视化；
- 数据分析准实时。

首先，分布式调用跟踪系统对生产环境中服务的性能损耗应该做到尽可能忽略不计，否则在一些特别注重性能的场景下，会严重影响系统整体的吞吐量，直接导致用户流失。其次，低侵入性也是一个非常重要的设计目标，分布式调用跟踪系统对于开发人员而言应该做到透明化，可以想象的是，在实际的开发过程中，业务团队每天只是应付大量的新增需求或者完善现有功能缺陷就已经应接不暇了，如果还需要在业务代码中进行大量的埋点上报工作，那么这样的监控系统未免也太脆弱了。再次，如图 1-14 所示，可视化的结果展现对于一个监控系统而言似乎是一个并不过分的要求，如果监控系统不具备可视化界面，那么后期推广时遇到的阻力可想而知。最后，数据的收集、运算和最终的结果展现应该做到快捷迅速，如果能够达到准实时级别的结果展现，开发人员就能够在服务异常的情况下及时做出反应和调整。

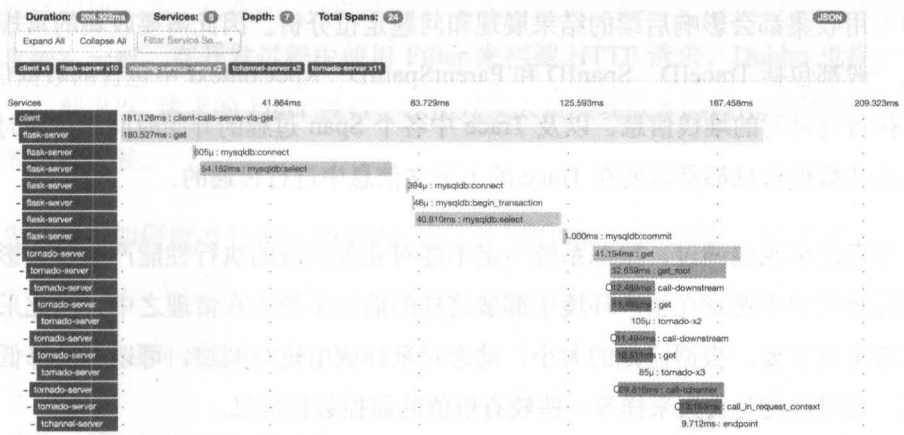


图 1-14　Twitter 的 Zipkin 系统监控界面

接下来我们来看看分布式调用跟踪系统的主要实现流程。

首先我们应该考虑的是如何将一次请求涉及的所有后端服务串联起来，这不仅是核心问题也是重点问题，只有将服务调用过程全部串联起来，才能够清楚一次请求的完整调用链。在这篇论文中，Trace 表示对一次请求的完整调用链追踪，而 Span 则可以理解为 Trace 的组成结构，比如服务 A 和服务 B 的请求/响应过程就是一次 Span。在生产环境中，一次用户请求可能涉及后端多个服务之间的调用，那么 Span 就用于体现服务之间具体的依赖关系。每一次请求都应该被分配一个全局唯一的 TraceID，并且整个调用链中所有的 Span 过程都应该获取到同一个 TraceID，以表示这些服务调用过程是发生在同一个 Trace 上的。

接下来我们来思考如何在服务或服务之间的调用过程中进行埋点和数据收集上报工作。一般来说，嵌入在 RPC 框架中进行埋点上报是最适合不过的（笔者选择嵌入到 Dubbo 框架中进行埋点和数据上报），因为这样就可以避免将埋点逻辑侵入业务代码中。而且，开发人员应重点关注的是自身的业务逻辑，而非埋点逻辑，甚至绝大多数开发人员并不愿意让这些非必需的耦合“污染”业务代码。埋点位置明确后，需要收集并上报哪些数据呢？由于上报的数据对后续的运算起了决定性作用，少收集和无用收集都会影响后续的结果展现和问题定位分析。因此需要收集的常规数据信息一般都包括 TraceID、SpanID 和 ParentSpanID、RpcContext 中包含的数据信息、服务执行异常时的堆栈信息，以及 Trace 中各个 Span 过程的开始时间和结束时间，并且这些数据信息都是需要在 Trace 的上下文信息中进行传递的。

笔者之前也强调过，跟踪系统一定不能对业务系统的执行性能产生较大影响，否则运维同学不愿意在生产环境中部署这样的监控系统也在情理之中，因此采样率便显得至关重要。根据流量的大小，动态对采样阈值进行调整，可以有效降低服务损耗，也可以避免丢失采样等一些较有价值的监控数据信息。

1.3.2 基于 Dubbo 实现分布式调用跟踪系统方案

笔者在前面为大家简单介绍了关于分布式调用跟踪系统的一些基础概念，接下来笔者就为大家演示如何基于 Dubbo 实现分布式调用跟踪系统。

Dubbo 预留了足够多的接口，开发人员可以非常方便地对其进行二次开发，在 Dubbo 框架上实现调用跟踪就显得顺理成章（Twitter 的 Zipkin 嵌入 Finagle 框架中，而淘宝的 EagleEye 则嵌入 HSF 框架中）。除此之外，嵌入在 Dubbo 中基本上可以做到对业务零侵入，不需要开发人员手动在业务代码中进行埋点上报工作，从而满足 Google 的 Dapper 论文的设计目标，对开发人员做到尽可能的透明化。

尽管市面上不同的服务调用跟踪产品在实现上或多或少都会存在一些差异，但是它们都具备以下两个基础功能：

- 跟踪每个请求的完整调用链；
- 收集调用链上每个服务的执行耗时，以及整合孤立日志。

既然是嵌入在 Dubbo 框架中实现服务的调用跟踪，那么方案已经明确了，但是具体应该如何实施，或者说，Dubbo 为开发人员提供了哪些扩展接口来实现调用跟踪需求呢？一般，在开发过程中使用 Filter 来拦截 HTTP 请求，Dubbo 也提供了专门用于拦截 RPC 请求的 Filter 接口，以便开发人员实现服务的调用跟踪和数据收集上报等功能扩展。

先来看看如何使用 Dubbo 的 Filter 接口对 RPC 请求进行拦截，如下所示：

```
/**
 * 实现一个用于拦截 RPC 请求的 Filter
 *
 * @author gaoxianglong
 */
```

```
public class Test_Filter implements Filter {
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation)
            throws RpcException {
        /* 前置拦截 */
        System.out.println("before");
        /* 执行目标服务调用 */
        Result result = invoker.invoke(invocation);
        /* 后置拦截 */
        System.out.println("after");
        return result;
    }
}
```

实现 Filter 接口后，开发人员还需要重写其 invoke(Invoker<?> invoker, Invocation invocation)方法。该方法中包含两个参数，其中 Invoker 接口提供用于执行目标服务方法的 invoke()方法，Invocation 接口可以向服务提供方传递当前 Trace 的上下文信息，其派生为 RpcInvocation 类。

成功编写好 Filter 后，还需要在 Spring 配置信息中对 Filter 进行配置，如下所示：

```
<!-- 服务调用方 Filter 配置 -->
<dubbo:consumer filter="testFilter" />
<!-- 服务提供方 Filter 配置 -->
<dubbo:provider filter="testFilter" />
```

在此需要注意，由于 Dubbo 的 Filter 并没有纳入 Spring 的 IOC 容器中进行管理，因此我们需要在/resources 目录下创建/META-INF/dubbo/com.alibaba.dubbo.rpc.Filter 文件，并在文件中通过键值对的方式指定 Filter 类的全限定名，如下所示：

```
testFilter=com.xx.xx.TestFilter
```

成功配置好 Filter 后，当服务调用方向服务提供方发起 RPC 请求时，Filter 将会对其进行拦截，开发人员便可以在远程服务方法的执行前后实现自定义的埋点上报逻辑。Dubbo 提供的 Filter 既可以对服务提供方进行拦截，也可以拦截服务调用方，只要在服务调用方和服务提供方的 Spring 配置信息中配置好 Filter 即可。

在 Filter 中可以使用 Dubbo 提供的一个临时状态记录器 RpcContext 类，通过调用 RpcContext 提供的一系列方法，可以非常方便地收集到当前 Span 过程中包含的一些非常有价值的数据信息，比如被 Filter 拦截的到底是服务调用方还是服务提供方、Host、Port、被调用的服务接口名称、被调用的服务方法名称，以及用户的一些自定义参数等。在分布式调用跟踪系统中，收集这些关键数据非常重要，因为如果没有这些数据信息作为支撑，那么后续跟踪系统将无法顺利进行数据运算和结果展现。使用 RpcContext 来获取当前 Span 过程的状态记录信息，如下所示：

```
RpcContext context = RpcContext.getContext();
if (null != context) {
    /* 是否是服务调用方 */
    context.isConsumerSide();
    /* 是否是服务提供方 */
    context.isProviderSide();
    /* 获取 Host 地址 */
    host = context.getLocalHost();
    /* 获取 Port */
    port = context.getLocalPort();
    /* 获取被调用的服务接口名称 */
    serviceName = context.getUrl().getServiceInterface();
    /* 获取被调用的服务方法名称 */
    methodName = context.getMethodName();
}
```

既然一次请求可能会涉及后端多个服务之间的调用，并且在并发环境下同一时

刻肯定会有许多用户请求进来，那么如何区分不同的服务调用究竟属于哪一个 Trace 呢？为 Trace 中的每一个 Span 过程都分配同一个全局唯一的 TraceID 是一个不错的方案，这样一来，一次请求中涉及的所有后端服务调用都会被完整地串联起来。那么 TraceID 应该如何生成呢？在单机环境中，生成一个全局唯一的 ID 似乎是一件非常简单的事情，但是在分布式环境中，生成一个既要考虑唯一性，又要兼顾连续性的 ID 就变得非常困难。关于在分布式环境中生成唯一 ID 的解决方案，大家可以直接阅读 5.2.6 节。

实现调用跟踪首先需要根据 TraceID 将一次请求中涉及的所有后端服务调用完整地串联起来形成一个 Trace，然后再考虑如何明确各个服务之间的调用顺序和依赖关系等问题。如图 1-15 所示，既然可以为 Trace 分配 TraceID，那么自然也可以为 Trace 中的每一个 Span 过程分配一个 SpanID 和 ParentSpanID。

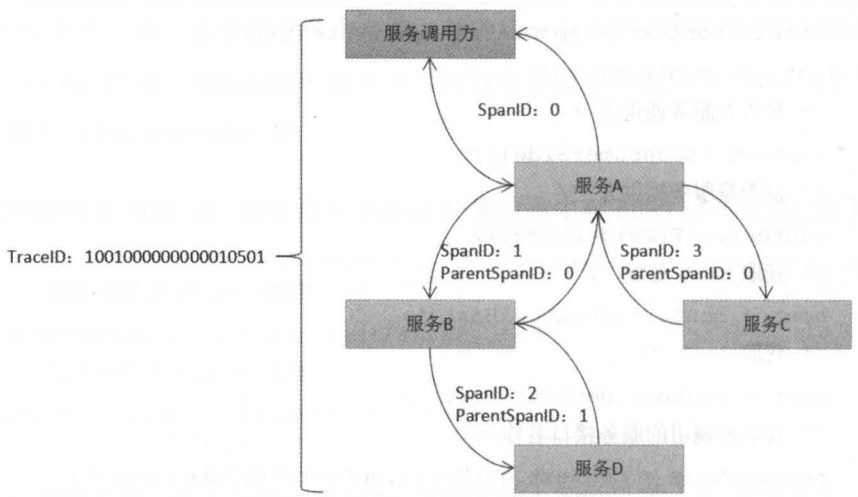


图 1-15 通过 TraceID 串联起来的 Trace

SpanID 用于标记每一个 Span 过程，代表着服务的调用顺序。而 ParentSpanID 则用于明确 Trace 中服务的依赖关系。SpanID 的值会随每一次服务调用递增，而

ParentSpanID 的值则来源于上一个 Span 过程的 SpanID，如下所示：

- Consumer 和服务 A 的请求/响应过程为一次 Span，由于是根调用，那么 SpanID 为 0，没有 ParentSpanID；
- 服务 A 和服务 B 的请求/响应过程为一次 Span，SpanID 为 1，ParentSpanID 为 0；
- 服务 B 和服务 D 的请求/响应过程为一次 Span，SpanID 为 2，ParentSpanID 为 1；
- 服务 A 和服务 C 的请求/响应过程为一次 Span，SpanID 为 3，ParentSpanID 为 0。

通过 TraceID、SpanID 和 ParentSpanID，便能够快速地梳理不同的 Trace 中服务之间的调用顺序和依赖关系，但是服务调用方如何将这些 Trace 上下文信息向下传递给服务提供方呢？如图 1-16 所示，当 Filter 对服务调用方进行拦截时，可以将 Trace 上下文信息 Set 进由 Dubbo 提供的 RpcInvocation 接口中；当 Filter 对服务提供方进行拦截时，再从中获取出之前由服务调用方传递过来的 Trace 上下文信息即可。关于 RpcInvocation 的使用，如下所示：

```
/**
 * 服务调用方传递给服务提供方的 Trace 上下文信息
 *
 * @author gaoxianglong
 */
public void setAnnotation(RpcInvocation rpcInvocation) {
    rpcInvocation.setAttachment("traceID","1001000000000010501");
    rpcInvocation.setAttachment("spanID", "1");
    rpcInvocation.setAttachment("parentSpanID", "0");
}

/**
```

```
 * 获取由服务调用方传递过来的 Trace 上下文信息
 *
 * @author gaoxianglong
 */
public void getAnnotation(RpcInvocation rpcInvocation) {
    rpcInvocation.getAttachment("traceID");
    rpcInvocation.getAttachment("spanID");
    rpcInvocation.getAttachment("parentSpanID");
}
```

图 1-16 在 RPC 请求中传递的 Trace 上下文信息

除了需要收集 TraceID、SpanID 和 ParentSpanID、RpcContext 中包含的数据信息，以及服务执行异常时的堆栈信息，还需要收集 Trace 中各个 Span 过程的开始时间和结束时间，以便跟踪系统后续根据这些收集到的时间参数进行服务的执行耗时运算。简单来说，当 Filter 拦截到 RPC 请求时，需要记录一个开始时间，当服务调用完成后还需要记录一个结束时间。那么服务调用方和服务提供方就是四个不同维度的时间戳，如下所示：

- Client Send Time（CS，客户端发送时间）；
- Client Receive Time（CR，客户端接收时间）；
- Server Receive Time（SR，服务端接收时间）；
- Server Send Time（SS，服务端发送时间）。

如图 1-17 所示，通过收集这四个不同维度的时间戳，便可以在一次请求完成后计算出整个 Trace 的执行耗时、网络耗时，以及 Trace 中每个 Span 过程的执行耗时、网络耗时等结果数据。关于服务执行耗时的运算规则，如下所示：

- 服务调用耗时=CR−CS；
- 服务处理耗时=SS−SR；
- 网络耗时=服务调用耗时−服务处理耗时；
- 前置网络耗时=SR−CS；
- 后置网络耗时=CR−SS。

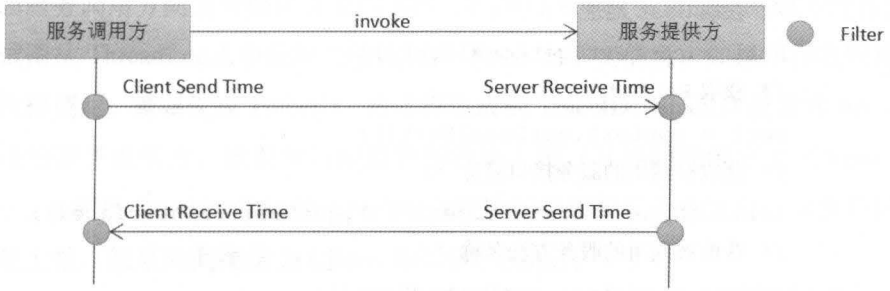


图 1-17 服务执行耗时的收集

或许有些同学会对服务调用耗时和服务处理耗时这两者之间的定义区分不清楚。简单来说，服务调用耗时表示一次 RPC 请求/响应所花费的总时间，而服务处理耗时则表示服务提供方执行服务方法所花费的时间，用前者减去后者就可以得出服务调用消耗在网络上的时间。网络耗时其实是一个非常重要的性能监控指标，因为在带宽紧张的情况下，网络访问速度的快慢将直接影响系统整体的吞吐量，当然网

络耗时我们还可以进一步细分为前置网络耗时和后置网络耗时。

在 Filter 中实现调用跟踪和数据收集上报的伪代码，如下所示：

```
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation)
    throws RpcException {
    /* 记录开始时间 */
    final long BEFORE_TIME = System.currentTimeMillis();
    RpcInvocation rpcInvocation = (RpcInvocation) invocation;
    /* 获取当前 Span 过程的状态记录信息 */
    RpcContext rpcContext = RpcContext.getContext();
    if (null != rpcContext) {
    /* 是否是服务调用方 */
        context.isConsumerSide();
        /* 是否是服务提供方 */
        context.isProviderSide();
        /* 获取 Host 地址 */
        host = context.getLocalHost();
        /* 获取 Port */
        port = context.getLocalPort();
        /* 获取被调用的服务接口名称 */
        serviceName = context.getUrl().getServiceInterface();
        /* 获取被调用的服务方法名称 */
    methodName = context.getMethodName();
    }
    /* 从 ThreadLocal 中获取当前线程的 Trace 上下文信息 */
    TraceBean traceBean = threadLocal.get();
    if (服务调用方) {
        if (ThreadLocal 中不包含当前线程的 Trace 上下文信息) {
            /* 根调用，生成 TraceID */
        } else {
```

```
            /* 递增 SpanID，设置 ParentSpanID */
        }
        /* 将 Trace 上下文信息设置到 Invocation 中 */
    }
    if (服务提供方) {
        /* 从 Invocation 中获取服务调用方传递过来的 Trace 上下文信息 */
         * 将 Trace 上下文信息设置在 ThreadLocal 中 */
       threadLocal.set(traceBean);
    }
    /* 前置数据收集上报 */
    beforeDataCollect(traceBean, BEFORE_TIME);
    Result result = invoker.invoke(rpcInvocation);
    /* 后置数据收集上报和一些收尾工作 */
    afterDataCollect(traceBean, System.currentTimeMillis());
    return result;
}
```

当服务调用方向服务提供方发起 RPC 请求时，Filter 会对服务调用方进行拦截，然后试图从 ThreadLocal 中获取当前线程的 Trace 上下文信息，如果不存在则说明这是一次根调用，需要生成 TraceID，然后将生成的 TraceID、SpanID 设置在 Invocation 中传递给服务提供方，接着执行前置数据收集上报（开始时间维度为 Client Send Time）。当调用 Invoker 接口的 invoke()方法执行完远程服务方法后，再执行后置数据收集上报（结束时间维度为 Client Receive Time）。

当 RPC 请求到达服务提供方后，Filter 会对其进行拦截，然后从 Invocation 中获取由服务调用方传递过来的 Trace 上下文信息，并将其存储到当前线程的 ThreadLocal 中，然后执行前置数据收集上报（开始时间维度为 Server Receive Time）。当调用 Invoker 接口的 invoke()方法执行完服务方法后，再执行后置数据收集上报（结束时间维度为 Server Send Time），最后还需要删除存储在 ThreadLocal 中当前线程的 Trace 上下文信息。

如果服务提供方内部还调用了其他服务，Filter 会对调用方进行拦截，然后从 ThreadLocal 中获取当前线程的 Trace 上下文信息，修改 SpanID 和设置 ParentSpanID 后，再将其设置到 Invocation 中传递给服务提供方，接着执行前置数据收集上报（开始时间维度为 Client Send Time）。当调用 Invoker 接口的 invoke()方法执行完远程服务方法后，再执行后置数据收集上报（结束时间维度为 Client Receive Time）。

图 1-18 所示为服务调用跟踪的大致流程。在此需要注意，如果将所有数据信息都直接写入数据库中将给数据库造成较大的负载压力，因此笔者建议将消息优先写入消息队列中，然后消费端消费到消息后，再写入数据库，以达到消峰效果。而且，底层存储除了可以使用关系型数据库，还可以尝试使用 HBase 之类的 NoSQL 数据库进行替代，当然这需要根据实际业务场景而定。如果上报的数据量不大，那么使用 MySQL 也未尝不可，并且由于监控数据的时效性较高，长期保存的意义不大，因此定期清理数据库中的历史数据也是非常必要的。

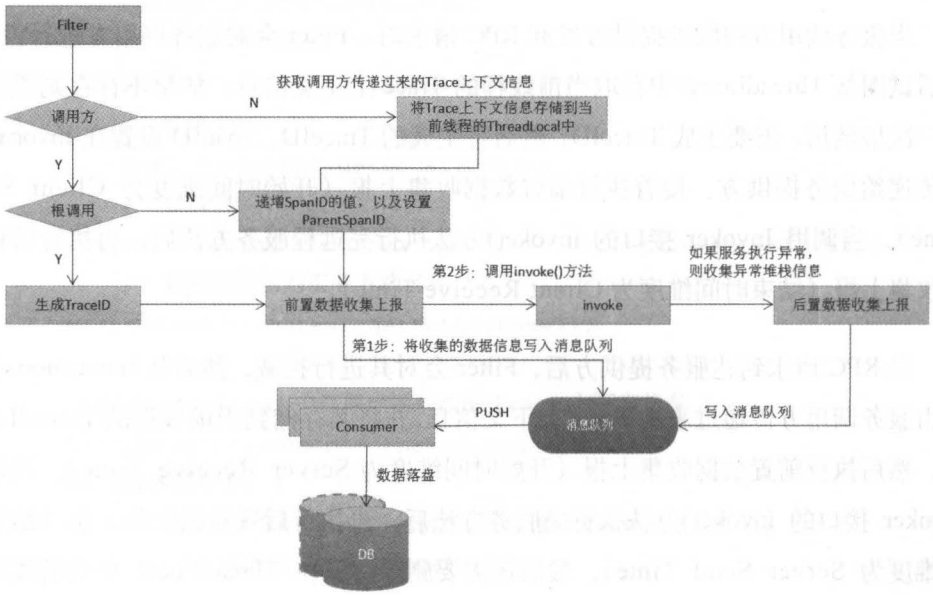


图 1-18 服务调用跟踪大致流程

关于后续分布式调用跟踪系统的数据运算和可视化结果展现等相关逻辑，本书就不再一一进行讲解了。如果大家感兴趣，可以从 GitHub 上下载本小节的完整示例代码进行阅读和扩展：https://github.com/gaoxianglong/service-tracing。

1.3.3 采样率方案

笔者在介绍 Google 的 Dapper 论文时，曾经提及过关于跟踪系统的四个关键设计目标，其中服务性能低损耗这个设计目标非常重要，因为在生产环境中如果跟踪系统对核心业务的性能影响较大，那么我们不得将其关停。因此架构师在设计和落地跟踪系统时，必然需要仔细权衡，跟踪系统所带来的收益一定要大于损耗服务性能的缺陷，否则运维人员可能不太愿意在生产环境中部署这种 Hold 不住的监控系统。

跟踪系统对服务的损耗主要在调用跟踪和数据收集上，除了开发人员自身的代码优化，还可以结合采样率在最大程度上控制损耗。在大促场景下，由于峰值流量较大，通常只需要采样其中很小的一部分请求即可（比如 1/1000 的采样率，跟踪系统只会在 1000 次请求中采样其中的某一次），从而降低跟踪系统给服务性能带来的损耗。采样控制最简单的做法是为所有的服务进程配置相同且固定的采样率，但这种做法显得极其不灵活，如果采样率阈值被设置得较低，用户流量较小就会错失很多重要的监控数据，因此笔者在生产环境中并没有采用固定的采样率配置，而是将采样率配置在配置中心内，以便运维人员根据实际的用户流量来对跟踪系统的采样率进行动态调整，这样既兼顾了服务性能又可以不丢失重要的监控数据，如图 1-19 所示。

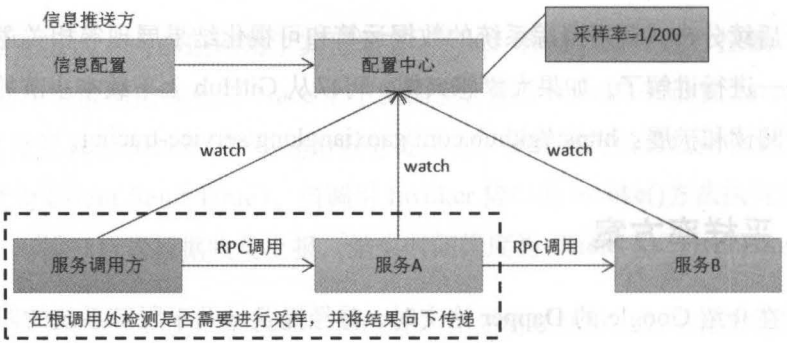


图 1-19 采样率设置和采样检测

由于将采样率配置在配置中心，一旦我们对采样率进行调整后，所有 Watch 到变化的服务进程便可以及时更新缓存在本地的采样率。当 Filter 拦截到 RPC 请求后，首先应该在根调用处判断是否需要对目标请求进行采样，然后再将判断结果设置在 Invocation 中向下传递给服务提供方。Filter 的采样检测伪代码，如下所示：

```
if (服务调用方) {
    if (根调用) {
        /* 判断是否对目标请求进行采样，并设置到 Invocation 中向下传递 */
    } else {
        /**
         *  从 ThreadLocal 中获取结果，判断是否需要进行采样
         *  并设置到 Invocation 中向下传递
         */
    }
}
if (服务提供方) {
    /**
     *  从 Invocation 中获取结果，判断是否需要进行采样
     *  并设置到 ThreadLocal 中
     */
}
```

1.4 本章小结

本章笔者为大家详细介绍了互联网领域分布式系统的架构演变过程。在此大家需要注意，如果用户规模及业务需求的复杂度还没有到量，那么最好保持现有架构不变，毕竟构建一个高性能、高可用、易扩展、可伸缩的分布式系统绝非一件简单的事情，需要解决的技术难题太多。而且，如果业务没有起色，一味地追寻大型网站架构并无任何意义。当然，随着用户规模的线性增长，以及业务需求越来越复杂，从单机系统逐渐演变为分布式系统，以更好地支撑业务发展似乎是必经之路。

拆系统是过渡到分布式系统的第一步，但是随着后续共享业务的增多、子系统水平扩展开始受限，以及技术团队规模开始膨胀，服务化改造势不可当。笔者以 RPC 协议作为切入点，为大家演示了阿里开源的服务化框架 Dubbo 的具体使用细节，以及如何通过 dubbo-admin 来实现服务治理。在本章的最后，笔者重点为大家演示了如何在 Dubbo 框架的基础之上实现一个分布式调用跟踪系统，以便于开发人员在生产环境服务异常的情况下及时做出反应和调整，为企业大规模服务化保驾护航。

2 第2章 大流量限流/消峰案例

天猫、淘宝这种级别的大型互联网电商网站，主要的技术挑战来自于庞大的用户规模所带来的大流量和高并发，在“双 11”、“双 12”等大促场景下尤为明显。如果不对流量进行合理管制，肆意放任大流量冲击系统，那么将导致一系列的问题出现，比如一些可用的连接资源被耗尽、分布式缓存的容量被撑爆、数据库吞吐量降低，最终必然会导致系统产生雪崩效应。当然，应对大流量和高并发也没有大家想象得那么复杂和神秘，一般来说，大型互联网站通常采用的做法是通过扩容、动静分离、缓存、服务降级及限流五种常规手段来保护系统的稳定运行。

不同的企业可能会采用不同的解决方案来应对大流量和高并发，因为不同业务之间存在的差异性决定了不会诞生通用的解决方案，所谓条条大路通罗马，只要能够换取同质性的结果，过程如何就显得不是特别重要，这就好比如果能够提前知道罗马的具体位置，那么处处都是路，怎么走都行。本章的重点是流量管制，只要我

们能够采用合理且有效的方式管制住用户流量，让其有秩序地对系统进行访问，那么无论是在平时还是在大促的时候，系统都能够稳定运行。需要明确的是，流量管制的目的是保护系统，让系统的负载处于一个比较均衡的水位，而不是刻意为了限流而限流，否则将会严重影响用户体验，得不偿失。

2.1 分布式系统为什么需要进行流量管制

在讨论系统为什么需要进行限流之前，我们先来聊一聊生活中那些随处可见的流量管制场景。笔者的居住地和工作地都在深圳，由于是一线城市，就以出行时乘坐地铁为例。在工作日的上下班高峰期，地铁站可谓人满为患，此期间地铁站的负载压力与春运相比简直是有过之而无不及，原本从站厅到站台最多只需花费 5 分钟左右的时间，却在地铁安保人员的流量管制下被迫花费 20～30 分钟才能够顺利进入站台，足足是平时的 5 倍多，其中的艰辛，相信挤过公交、地铁的同学应该都能够感同身受。

那么，为什么在流量管制后需要花费这么长的时间才可以顺利搭乘地铁呢？其实就是排队，从站厅层开始，工作人员会慢慢引流，这样站台层的压力就会减小很多，不会使大量无秩序的乘客全部拥堵在站台层，导致想上的人上不去，想下的人下不来，车门关不上，地铁也无法顺利行驶（笔者经常看见一些乘客不顾生命危险在地铁门关上的一刹那冲抢车门，导致背包、外套，以及头发被安全门夹住等悲剧发生），还会导致后续列车因为临时停车而影响到站时间，因此牺牲一点个人时间换来整体的井然有序是非常值得的，如图 2-1 所示。试想一下，如果早晚高峰期地铁不实施流量管制，那么一定会导致站厅层和站台层都被挤得水泄不通，就算你使出洪荒之力，也不一定就能够挤得上车，就算你侥幸挤上去了，也一定会被挤压成陕西名吃“肉夹馍”，并且在人满为患的公共场所，最怕的就是拥挤，因为一旦拥挤就非常容易发生安全事故，所以限流，既能够保证每一位乘客最终都能够顺利上车，又能够确保地铁行驶有条不紊地进行。

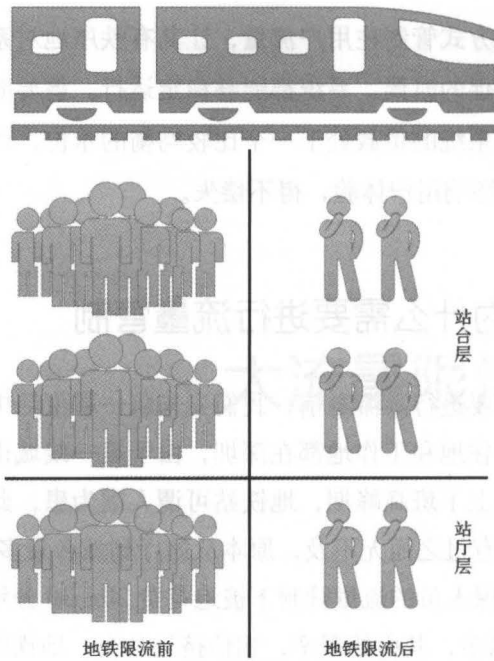


图 2-1 地铁高峰期流量管制

2016 年的“双 11”大促活动，在剁手党们的齐心协力下，天猫商城又一次刷新了历史，52 秒 10 亿元、不到 7 分钟破百亿元，最终以 1207 亿元的总成交记录傲视群雄，简直堪称行业奇迹。大家思考一下，在傲人的成绩面前，阿里的技术团队是依靠哪些“高精尖”的技术才让系统有效扛住如此庞大的用户流量呢？简单来说，大型网站的成长往往都伴随着大流量和海量数据的洗礼，那么在类似“双 11”这种大促场景下，笔者总结了以下五种分布式系统应对高并发、大流量的常规手段：

- 扩容；
- 动静分离；
- 缓存；
- 服务降级；
- 限流。

由于单台服务器的处理能力有限，因此当一台服务器的处理能力接近或已超出其容量上限时，采用集群技术对服务器进行扩容，可以很好地提升系统整体的并行处理能力，在集群环境中，节点的数量越多，系统的并行处理能力和容错性就越强。动静分离其实是一个老生常谈的话题，简而言之，系统需要将动态数据和静态数据分而治之，用户对静态数据的访问，应该避免请求直接落到企业的数据中心，而是应该在 CDN 中获取，以加速系统的响应速度。大促场景下热点数据的读/写操作一直就是最核心的技术难题，通过缓存技术，系统在应对高并发、大流量时可谓如虎添翼，因为缓存的读/写效率要远胜于任何关系型数据库，合理地使用缓存技术，系统的吞吐量将会得到质的提升。在 1.2.4 节中，笔者为大家介绍了大规模服务化场景下服务治理的重要性，其中服务的升/降级处理尤为重要，当系统容量支撑核心业务都捉襟见肘时，牺牲掉部分功能换来系统的核心服务不受影响是非常有必要的，毕竟有损服务和系统宕机完全不能对外提供服务是两码事。和现实生活中流量管制的场景类似，当网站举行大促活动时，那些单价比平时更给力、更具吸引力的热卖商品一定会吸引大量的用户前来购买，因此需要采用合理且有效的限流手段对系统做好保护，毕竟不是任何场景都可以仅通过缓存和服务降级等技术就能够实现一本万利，如系统中的写服务（用户下单、库存扣减、商品评论等）。

任何一个分布式系统的容量都会存在上限，哪怕天猫这种级别的网站也不例外。一旦用户流量过载，系统的吞吐量便会开始下降，RT 线性上升，最终导致系统容量被撑爆而出现雪崩效应。因此，架构师在对系统架构进行设计时，一定要考虑到系统整个链路中的各个环节，就像笔者所示应对高并发、大流量的五种常规手段一样，这些看似平淡无奇甚至“毫无新意”的技术，组合在一起时却能爆发出惊人的力量。在此需要注意，对于大型网站而言，其架构一定是简单和清晰的，而不是炫技般的复杂化，毕竟解决问题采用最直接的方式直击要害才是最见效的，否则事情只会变得越来越糟。

如图 2-2 所示，在高并发、大流量场景下合理地运用扩容、动静分离、缓存、服

务降级及限流五种常规手段，可以使用户流量像漏斗模型一样逐层减少，让流量始终保持在系统可处理的容量范围之内。

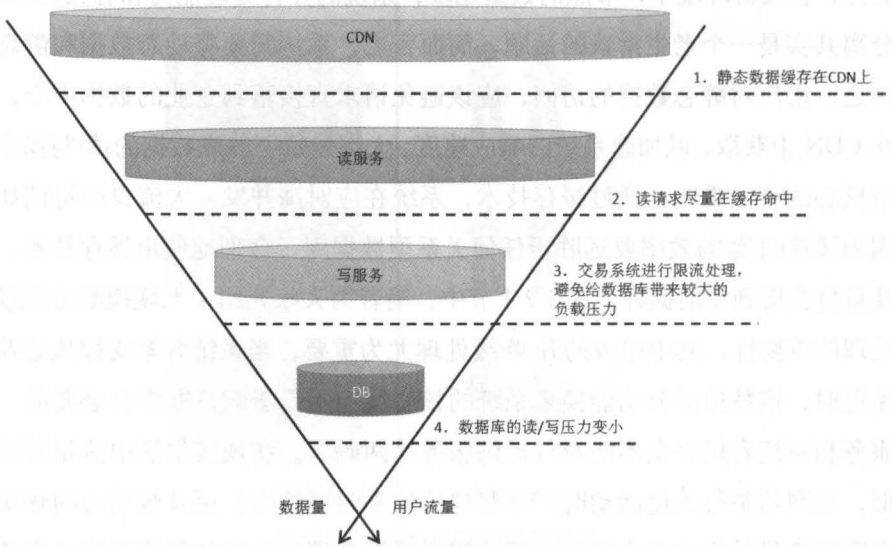


图 2-2 漏斗模型

2.2 限流的具体方案

限流的手段是非常多样化的，不同的互联网企业会因为业务的不同而选择不同的限流技术、限流算法，所以就限流本身的选型而言，业界并没有统一的标准，只要系统能够通过某种限流手段合理地对流量实施管制，保证系统能够有条不紊地运行，那么使用哪一种限流技术就不是特别重要。值得一提的是，大家千万不要盲目地用所谓的大公司的解决方案给自己套上枷锁，并不是说互联网巨头用什么技术或者解决方案大家就要一味地照搬不误，只有适用于自身业务的才是最合适的，否则被带沟里去了还全然不知。

笔者在实际工作中，对各种限流技术都有幸接触过并使用到了，因此笔者接下来便会针对这些不同类型的限流技术的具体应用场景进行重点讲解。

2.2.1 常见的限流算法

可以毫不客气地说，在实际的开发过程中，基本上每个开发人员都直接或间接地与限流算法打过交道，比如池化资源技术（数据库连接池、线程池、对象池等）。以数据库为例，我们都知道数据库连接是一种非常昂贵且数量有限的底层资源，为了避免并发环境下连接数超过数据库所能够承载的最大上限，合理地运用连接池技术，可以有效限制单个进程内能够申请到的最大连接数，确保在并发环境下连接数不会超过资源阈值。池化资源技术的限流其实就是通过计数器算法来控制全局的总并发数，笔者在生产环境中，正是使用计数器算法来实现大促场景下的商品抢购限流的。关于抢购限流的解决方案，大家可以直接阅读 2.2.4 节。除了基于计数器的限流算法，目前市面上还有以下两种比较常见的限流算法。

1. 令牌桶算法

令牌桶（Token Bucket）算法主要用于限制流量的平均流入速率，并且还允许出现一定程度上的突发流量，如图 2-3 所示。基于令牌桶算法的限流场景较多，比如 Nginx 的限流模块就是一个典型的采用令牌桶算法的实现。令牌桶算法限流的基本流程如下所示：

- 每秒会有 r 个令牌被放入桶内，也就是说，会以 $1/r$ 秒的平均速率向桶中依次放入令牌（比如每秒共放入 10 个令牌，那么每 0.1 秒放入 1 个令牌）；
- 桶的容量是固定不变的，假设桶中最多只允许存放 b 个令牌，如果桶满了再放入令牌，则溢出（新添加的令牌被丢弃）；
- 当一个 n 字节的请求包到达时，将消耗 n 个令牌，然后再发送该数据包；
- 若桶中的可用令牌数小于 n，则该数据包将会被执行限流处理（被抛弃或缓存）。

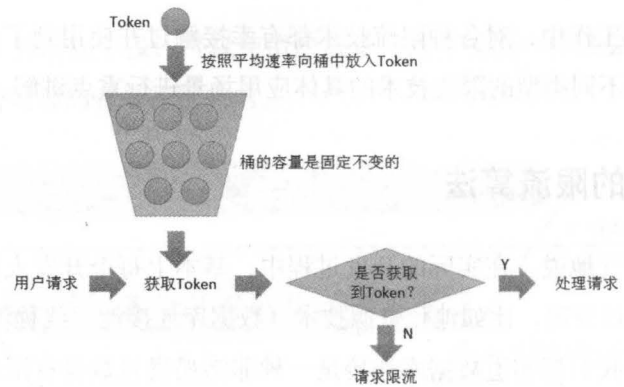


图 2-3 令牌桶算法流程

2. 漏桶算法

漏桶算法（Leaky Bucket）也可以高效地实现流量管制，如图 2-4 所示。漏桶算法限流的基本流程如下所示：

- 可以以任意速率向桶中流入水滴；
- 桶的容量是固定不变的，如果桶满了则溢出（新流入的水滴被丢弃）；
- 按照固定的速率从桶中流出水滴。

图 2-4 漏桶算法流程

从本质上来说，令牌桶算法和漏桶算法都可以用于在高并发、大流量场景下对流量实施管制，让系统的负载处于比较均衡的水位，不会因为峰值流量过大，导致系统被击垮。但是需要注意，这两种算法的限流方向是截然相反的。令牌桶算法限制的是流量的平均流入速率，并且可以允许出现一定程度上的突发流量，当桶中令牌数量不足扣减时，新的请求将被执行限流处理；而漏桶算法限制的是流量的流出速率，而不是流入速率，并且这种流出速率还是保持固定不变的，不允许像令牌桶算法那样出现突发流量，当流入的水滴超过桶的容量时，新的请求将被执行限流处理。

2.2.2 使用 Google 的 Guava 实现平均速率限流

和 Apache 的 Commons 类似，Guava 是 Google 提供的对 Java API 进行上层封装的开源工具包，在开发过程中使用 Guava 或 Commons 提供的类库，可以大幅度减少开发人员的编码量，使开发人员更专注于自身的业务逻辑，提升开发效率。Guava 最大的特点就是封装了 Java API 中的集合框架和 Cache 等特性。因此，在本地缓存领域除了可以使用 EhCache，Guava Cache 也是一个非常不错的替代方案。当然，笔者并不打算对 Guava 的所有特性都逐一进行讲解，仅仅针对 Guava 提供的限流功能进行演示。如果大家感兴趣，可以自行下载 Guava 的源码进行阅读和分析，网址为 https://github.com/google/guava.git。

如果希望在程序中实现基于令牌桶算法那样的平均速率限流，那么使用 Guava 的 RateLimiter 抽象类会是一个不错的选择。简而言之，RateLimiter 提供了令牌桶算法的实现，使得我们可以非常方便地在程序中实现流量的平均流入速率限流。

本书所使用的 Guava 版本为 18.0，大家可以通过 Maven 依赖的方式下载 Guava 的相关构件，如下所示：

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>18.0</version>
</dependency>
```

成功下载好运行 Guava 所需的相关构件后，就来看看应该如何使用 RateLimiter 提供的限流功能，如下所示：

```
public void limit() {
    /* 每秒向桶中存放 5 个 Token */
    final RateLimiter limit = RateLimiter.create(5);
    for (int i = 0; i < 10; i++) {
        /* 从桶中获取 1 个 Token */
        double waitTime = limit.acquire();
        System.out.println(waitTime);
    }
}
```

在上述程序示例中，笔者通过调用 RateLimiter 的 create(double permitsPerSecond) 方法创建了一个 RateLimiter 实例，参数“permitsPerSecond”设置了每秒向桶中放入 5 个 Token。假设每次每个请求都只从桶中获取 1 个 Token，那么大约需要 2 秒才可以从桶中获取 10 个 Token，也就是说，系统每秒只允许 5 个并发请求。输出结果如下所示：

```
0.0
0.199448
0.200018
0.200204
0.200203
0.200184
```

```
0.200183
0.200209
0.200172
0.200144
```

为了应对突发流量，RateLimiter 也支持请求一次从桶中获取多个 Token，如下所示：

```
public void limit() {
    /* 每秒中向桶中存放 5 个 Token */
    RateLimiter limit = RateLimiter.create(5);
    /* 产生突发流量时，一次从桶中获取 5 个 Token */
    System.out.println(limit.acquire(5));
    System.out.println(limit.acquire());
}
```

在上述程序示例中，笔者模拟了一次突发流量的情况，当请求一次性拿完了桶中仅有的5个Token后，大约需要等待1秒，后续请求才可以继续从桶中获取出Token，因为每 0.2 秒放入 1 个 Token，需要在偿还完之前因突发流量提前透支的 Token 数量后，才能允许请求进行消费。假设在 limit.acquire(5)方法前休眠 1 秒，限流效果将发生变化，因为程序在休眠时，桶中已经被放入了足够数量的 Token。

除了标准的平均速率限流，如果希望系统在启动时的限流速率从慢速逐渐过渡到平均速率，给系统一个缓冲时间，那么 RateLimiter 也提供有相应的支持，只需要在 create()方法中设置缓冲时间即可，如下所示：

```
RateLimiter.create(5, 1, TimeUnit.SECONDS);
```

在 create(double permitsPerSecond, long warmupPeriod, TimeUnit unit)方法中，参数“warmupPeriod”用于设置限流速率从慢速过渡到平均速率的缓冲时间，参数“unit”

则用于设置缓冲时间单位。在此需要注意，假设请求被执行限流后，我们不希望请求一直处于等待状态获取 Token，而是直接丢弃或短暂等待，那么便需要使用 RateLimiter 提供的 tryAcquire()方法，如下所示：

```
/* 尝试从桶中获取 Token，获取不到不等待立即返回 false */
boolean result = limit.tryAcquire();
if (result) {
    System.out.println("成功获取到 Token");
}
/* 尝试从桶中获取 Token，只等待 10ms */
result = limit.tryAcquire(10, TimeUnit.MILLISECONDS);
if (result) {
    System.out.println("成功获取到 Token");
}
```

2.2.3 使用 Nginx 实现接入层限流

Nginx 是一个开源的高性能 HTTP 服务器，同时也可以作为一个反向代理服务器，甚至还可以作为一个 IMAP/POP 3/SMTP 服务器。由于 Nginx 拥有强悍的并发处理能力，因此许多互联网企业都将 Nginx 部署在接入层，将其当作反向代理服务器来使用，负责请求的负载均衡和分发等工作。鉴于 Nginx 的优异性能，阿里开源的 Tengine 产品就是在 Nginx 的基础上进行二次开发的，并专门针对高并发、大流量场景进行了诸多优化，还提供了一些 Nginx 所没有的高级功能和特性。如果大家感兴趣，可以自行下载 Tengine 的源码进行阅读和分析：https://github.com/alibaba/tengine。

除了可以使用 Nginx 负责请求的负载均衡和分发等工作，Nginx 自带的限流模块还可以有效帮助运维人员在接入层中限制流量的平均速率。

开启 Nginx 的限流功能，如下所示：

```
http{
    # 定义每个 IP 的 session 空间大小
    limit_zone one  $binary_remote_addr  20m;
    # 与 limit_zone 类似，定义每个 IP 每秒允许发起的请求数
    limit_req_zone  $binary_remote_addr  zone=req_one:20m rate=10r/s;
    # 定义每个 IP 能够发起的并发连接数
    limit_conn   one  10;
    # 缓存还没来得及处理的请求
    limit_req    zone=req_one burst=100;
server{
        listen          80;
        server_name     localhost;
        location / {
            stub_status  on;
            access_log   off;
    }
    }
}
```

在上述 Nginx 的配置信息中，参数 limit_req_zone 用于设置每个 IP 在单位时间内所允许发起的请求数，值“rate=10r/s”表示每个 IP 每秒只允许发起 10 个请求。如果每秒每个 IP 发起的请求数超过 10 个应该怎么办呢？参数 limit_req 的作用类似于缓冲区，用于缓存还没有来得及进行处理的请求，值“burst=100”表示缓存的请求数为 100，如果缓冲区中也缓存不下时，新的请求将会被拒绝和抛弃。

2.2.4 使用计数器算法实现商品抢购限流

除了令牌桶、漏桶等限流算法，计数器算法也可以达到不错的限流效果。笔者

以限时抢购场景为例，当用户成功下单后便需要扣减指定商品的库存数量，但是大量的并发请求对数据库中同一行记录执行更新操作必然会导致数据库出现较大的负载压力。关于热点数据的读/写优化案例，大家可以直接阅读第4章。在此需要注意，针对瓶颈点进行优化是一方面，如果系统前端能够配合交易系统做好限流保护更可事半功倍。

那么在程序中我们应该如何实现商品抢购限流呢？简单来说，抢购限流其实指的就是指定的SKU在单位时间内允许被抢购的次数，一旦超过所设定的阈值，那么系统将会拒绝后续用户的抢购请求（如果希望拥有较好的用户体验，客户端可以配合实施抢购失败时的排队等待页面效果）。比如，某一个SKU的限流规则为10秒5000次，如果10秒之内这个SKU的抢购次数达到了5000次，则拒绝后续抢购请求，10秒之后，限流逻辑可以对这个SKU的可抢购次数（计数器）进行重置，确保之前没有抢购成功的用户可以继续正常参与抢购。抢购限流的伪代码如下所示：

```
public boolean limit(List<String> skus) {
    if (检测限流开关是否打开) {
        if (null != skus && !skus.isEmpty()) {
       /* key为SKU，value为可抢购的剩余次数 */
            Map<String, Integer> skuMap = 获取被限流的SKU名单;
            synchronized (skuMap) {
                try {
                    for (String sku : skus) {
                        /* 检查目标SKU是否包含在限流名单中 */
                        if (skuMap.containsKey(sku)) {
                            while_skus.add(sku);
                            /* 如果抢购次数达到阈值，则不允许下单 */
                            if (0 >= skuMap.get(sku))
                                return false;
                        }
                    }
```

```
                    for (String sku : while_skus) {
                        /* 扣减可抢购次数 */
                        skuMap.put(sku, skuMap.get(sku) - 1);
                    }
                } finally {
                    if (!while_skus.isEmpty()) {
                        while_skus.clear();
                    }
                }
            }
        }
    }
    return true;
}
```

在生产环境中，笔者将限流规则配置在配置中心内，以便后续对限流开关进行动态关停和变更不同 SKU 的限流规则。当然，可抢购次数的扣减操作，既可以在 Redis 等集中式容器中进行，也可以直接在本地进行。无论选择哪一种方式，限流效果均是一致的，只是后者在实现上会更加简单。

假设商品业务是一个由 10 台机器构成的集群，并且希望指定的 SKU 在 10 秒内只允许被抢购 5000 次，如果在本地扣减可抢购次数，那么限流规则应该被设定为 10 秒 500 次（可抢购次数/节点数），这就很好地控制住了单机的并发写流量。

2.3 基于时间分片的消峰方案

笔者在前面几个小节中曾为大家介绍了如何通过技术手段进行流量管制，其实，也可以在业务上做调整。大家千万要记住，试图采用技术来解决一切问题是非常愚蠢的，通常我们使用技术是为了实现特定的业务需求，但如果业务本身存在问题（如

12306 网站早期的业务架构），那么架构师就应该尝试换个思路或玩法，只要系统能够有效 Hold 住流量，保证不被无情地击垮，便是赢。

所谓消峰，其实指的就是对峰值流量进行分散处理，避免在同一时间段内产生较大的用户流量冲击系统，从而降低系统的负载压力。接下来笔者就为大家讲解如何在业务上做调整来实现流量消峰。

2.3.1 活动分时段进行实现消峰

大促活动整点的波峰值流量如同一把尖锐的刺刀，让人不寒而栗，但是如果将整点的促销活动调整到多个时段进行（比如某一个 SKU 的库存数量为 5000，抢购时段被分为 10 次，那么运营人员在每个时段放置的库存数量为 500（SKU/时段）），同一时段聚集的用户流量将会被有效分散，大家都不会火急火燎地在同一个时间点去抢购心仪的爆款商品，这样系统的负载压力将会大大降低。

在业务上做调整来对流量进行消峰，也能够收获非常好的限流效果，这便是站在业务的角度对系统实施保护的一个非常典型的案例。

2.3.2 通过答题验证实现消峰

基于时间分片进行消峰的玩法还是比较丰富的，笔者认为其中最有趣的当属答题验证。参与过抢购的同学对于答题验证环节应该不会感觉到陌生，在活动开始前，我们就盯着显示器上心仪的商品希望能够在活动开始后顺利将其收入囊中，但很多时候却在答题验证环节出了错误，最终导致抢购失败。

随着机器学习的日趋成熟，一些简单的字符、算数运算、图片等多种形式的验证码显然已经无法有效阻挡秒杀器（通过光学字符识别，或者智能图像识别等技术可以快速准确地进行识别）的步伐，因此提升验证码难度和降低图片画质就迫在眉

睫。笔者认为中国铁道部的 12306 网站将验证码的难度提升到了一个新的高度，在一票难求的春运抢票活动中，尽管那一道道各式各样的“奇葩”验证码，遭到了广大“吃瓜群众”的集体吐槽，但却成功阻挡了秒杀器和降低了峰值流量，如图 2-5 所示。

图 2-5 12306 网站的验证码

如果用户在下单前需要经历答题验证环节，那么峰值的下单请求必然会被拉长，并且靠后的请求自然也会因为没有库存而无法顺利完成下单，因此同一时间对系统进行并发写的流量将会非常有限。

2.4 异步调用需求

面向对象（Object Oriented）的本质是让代码能够更好地实现复用，但是任何基于面向对象的编程语言，似乎都摆脱不了软件工程中一个永恒的问题，那便是代码如何才能够更好地实现解耦。尽管并不存在最优的解耦方案，但是只要架构师和开发人员能够在系统的设计和实现上遵循单一化原则，尽量面向接口编程，以及善用设计模式，那么耦合自然就降低了。当然，如果想在程序中完全解除耦合是不切实际的，因为如果程序中没有了耦合，我们将什么事也做不了。

异步调用是能够显著提升程序执行性能的技术，开发人员可以通过创建线程来

实现方法的异步调用，将程序中原本的串行化执行流程变为并发/并行执行，并且在 Java 7 的新增语法特性中，Fork/Join 框架的到来也正式让 Java 语言过渡到了多核并行计算时代，由于能够将一个任务量化到最小，并提供了高计算密度的并行处理能力，因此任务的执行效率将得到质的提升。看到这里或许有些人已经产生了疑问，异步调用和系统解耦之间究竟存在什么必然的联系呢？在第 1 章中，笔者曾为大家讲解过如何通过业务垂直化、系统独立部署等分而治之的手段来实现不同业务、不同系统之间的解耦。除此之外，通过消息中间件实现异步调用也是在分布式环境下解决系统之间的耦合及大流量消峰的重要手段。

2.4.1 使用 MQ 实现系统之间的解耦

在本书示例的很多技术难题的解决方案中，大家都会看到 MQ（Message Queue，消息队列）的身影，在大部分情况下，我们在程序中使用 MQ 是为了实现特定场景下的流量消峰，以及通过消息传递来实现异步调用等操作。由于 MQ 技术发展至今已经相当成熟了，目前市面上也汇集了许多优秀的开源 MQ 产品，如 Apache 的 ActiveMQ 和 Kafka、阿里的 RocketMQ，以及 HornetQ、RabbitMQ、ZeroMQ 等，笔者建议用户规模较小的网站使用遵循 JMS（Java Message Service）规范的 ActiveMQ 这种轻量级的 MQ 产品（甚至也可以使用 Redis 提供的 Publish/Subscribe 模型），而 KafKa、RocketMQ 等类型的 MQ 产品，天生就是为互联网场景下拥有高并发、大流量的分布式系统量身打造的。因为在这种规模的消息体量上，我们重点需要考虑的是 MQ 产品的吞吐量、可用性及扩展性等多个方面的问题。

关于如何使用 MQ 来进行大流量消峰，大家可以直接阅读 2.4.4 节。当然，在此之前，笔者首先为大家讲解在程序中应该如何使用 MQ 实现系统之间的解耦。当业务垂直化并独立部署后，尽管不同的业务子系统之间可以通过 RPC 请求来实现服务调用，但是在某些情况下，这些依赖又并不一定都是必需的，因此完全可以使用消息传递来替代 RPC 调用。比如，某个用户向用户系统发起注册请求，假设注册操作

需要经历注册校验、新用户信息入库、调用邮件系统发送账号激活邮件、激活新账号四个步骤，但是对于用户系统而言，邮件系统却并不是必须依赖的，也就是说，当完成注册步骤后，我们可以立即选择将消息写入到消息队列中，待消费者消费后，异步完成激活邮件的发送即可，如图 2-6 所示。如果用户系统中包含较多的非必需依赖，那么必然会提升其复杂度和维护成本。总之，那些非必需的依赖，在实际的开发过程中我们都可以通过消息传递来进行替代，从而保证进程功能的单一性。

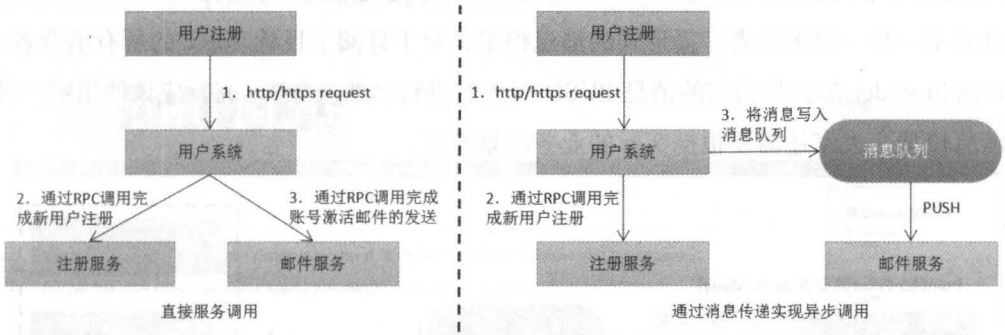


图 2-6 通过消息传递实现异步调用

2.4.2 使用 Apache 开源的 ActiveMQ 实现异步调用

既然谈到 ActiveMQ，就不得不提及 JMS 规范，毕竟 ActiveMQ 是一个遵循了 JMS 规范的开源消息中间件实现。根据 JMS 的架构模型来看，JMS 主要由 JMS Provider、Provider 及 Consumer 三个角色构成，其中 JMS Provider 的主要任务是负责消息路由和消息传递，而 Provider（消息生产者）和 Consumer（消息消费者）其实都属于 JMS 客户端，只是前者负责向消息队列写入消息，而后者负责订阅消息。JMS 的消息模型有两种，如下所示：

- Point-to-Point（P2P，点对点）模型；
- Publish/Subscribe（pub/sub，发布/订阅）模型。

如图 2-7 所示，JMS 两种类型的消息模型其实都非常简单和易于理解，Point-to-Point 实际上就是一对一的消息推送/消费模式。简单来说，如果有多个消费者都在监听消息队列上的消息，那么 JMS Provider 则会根据先到先得的原则确定唯一的消费者，然后由指定的消费者对目标消息进行消费，当然如果目前还没有任何的消费者在监听消息队列，那么未被消费的消息将会被暂时积压在消息队列中，直至最终被消费为止。实际上，Point-to-Point 是一个典型的 PULL 模式，由消费者主动从消息队列中获取消息，这和 Publish/Subscribe 模型是截然不同的。Publish/Subscribe 其实是一种一对多或者广播形式的消息模型，对于订阅了目标 Topic 的所有消费者，JMS Provider 都会将指定的消息 PUSH 给他们进行消费。当然，究竟应该使用哪一种消息模型，大家还需要根据实际的业务场景而定。

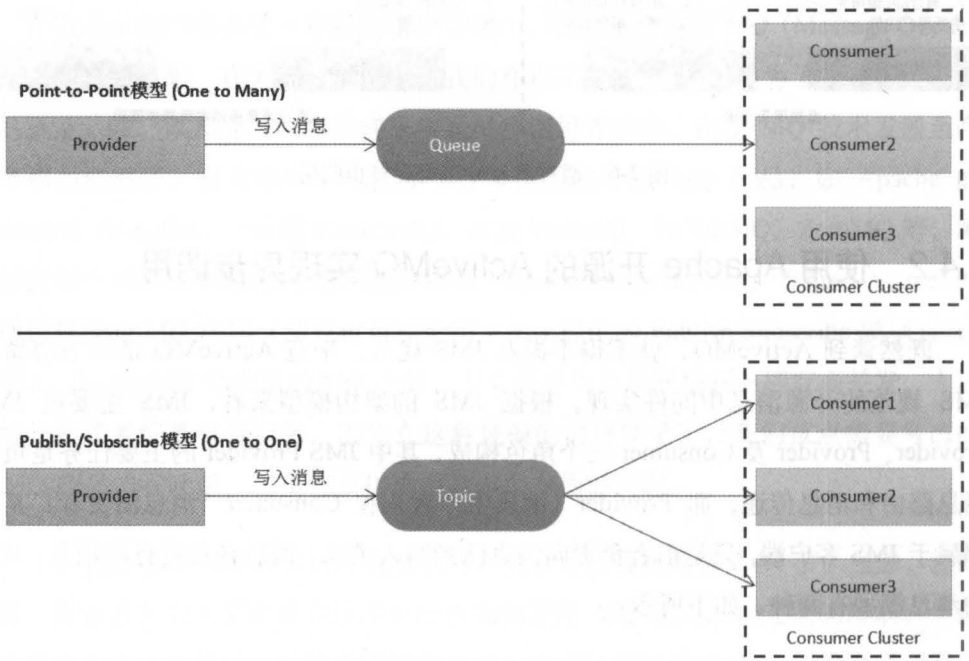


图 2-7 JMS 的两种消息模型

当大家对 JMS 规范有所了解后，笔者就为大家演示如何在程序中使用 ActiveMQ 来进行消息传递。ActiveMQ 分为客户端和服务端，客户端对应的是消息的生产者和消费者，而服务端对应的则是 JMS 规范中的 JMS Provider 角色。大家可以从 http://activemq.apache.org/download.html 处进行下载，然后启动消息服务端。成功启动消息服务后的界面如图 2-8 所示，大家可以直接访问 ActiveMQ 提供的管理控制台界面（http://host:8161/admin），其初始账号密码为 admin/admin，如果想修改初始的账号和密码，直接编辑/conf/jetty-realm.properties 文件即可。

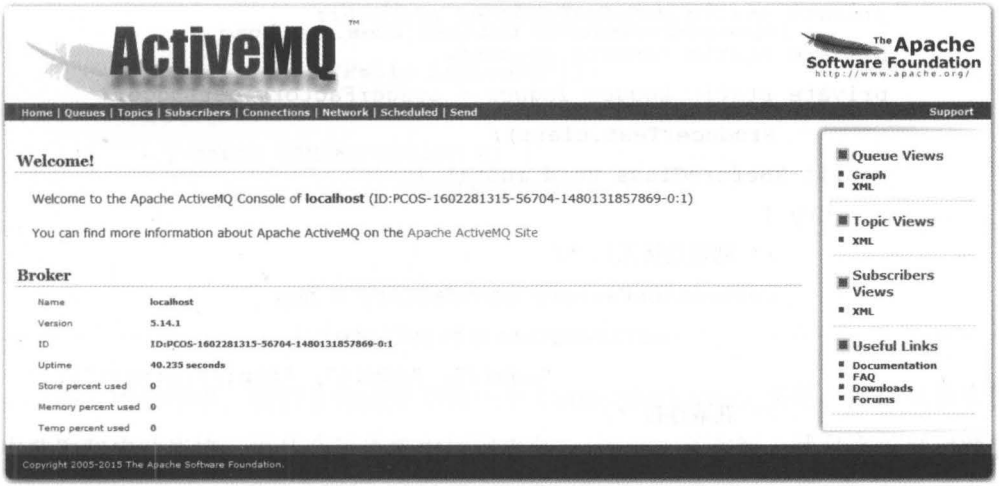


图 2-8 ActiveMQ 的管理控制台界面

本书使用的 ActiveMQ 版本为 5.14.1，开发人员可以通过 Maven 依赖的方式下载 ActiveMQ 构件，如下所示：

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-client</artifactId>
    <version>5.10.0</version>
</dependency>
```

成功下载好运行 ActiveMQ 所需的相关构件后，首先要做的事情就是编写消息生产者，如下所示：

```
/**
 * 基于 ActiveMQ 的消息生产者
 *
 * @author gaoxianglong
 */
public class ProducerTest {
    private static MessageProducer producer;
    private static Session session;
    private static Logger logger = LoggerFactory.getLogger(
            ProducerTest.class);
    public @BeforeClass void init() {
        try {
            /* 创建连接工厂 */
            ConnectionFactory connFactory = new
                    ActiveMQConnectionFactory(
                            "admin", "admin", "tcp://ip:port");
            /* 获取连接 */
            Connection conn = connFactory.createConnection();
            conn.start();
            /* 获取事务 session */
            session = conn.createSession(true,
                    Session.AUTO_ACKNOWLEDGE);
            /* 创建基于 Point-to-Point 模型的生产者 */
            producer = session.createProducer(
                    session.createQueue("testQueue"));
            producer.setDeliveryMode(
                    DeliveryMode.NON_PERSISTENT);
        } catch (JMSException e) {
            try {
```

```
                /* 事务回滚操作 */
                session.rollback();
            } catch (JMSException e1) {
                e1.printStackTrace();
            }
        }
    }
    public @Test void sendMessage() {
        try {
            /* 向消息队列写入消息 */
            producer.send(session.createTextMessage(
                    "Hello ActiveMQ"));
            session.commit();
        } catch (JMSException e) {
            logger.error("消息推送失败",e);
        }
    }
}
```

在上述程序中，我们首先需要创建一个 ConnectionFactory 实例，然后从该实例中获取和启动连接，再从会话连接中创建事务 session，并在写入消息之前创建消息生产者和设置消息模型（Session 的 createQueue(String queueName)方法用于设置 Point-to-Point 模型，而 createTopic(String queueName)方法用于设置 Publish/Subscribe 模型），当一切准备就绪后，调用 MessageProducer 的 send(Message message)方法即可成功将消息写入到消息队列中。

接下来我们来看看如何编写消费者代码，如下所示：

```
/**
 * 基于 ActiveMQ 的消息消费者
 *
```

```
 * @author gaoxianglong
 */
public class ConsumerTest implements MessageListener {
    private static MessageConsumer consumer;
    private static Logger logger = LoggerFactory.getLogger(
            ConsumerTest.class);
    private @BeforeClass void init() {
        try {
            /* 创建连接工厂 */
            ConnectionFactory connFactory = new
                    ActiveMQConnectionFactory(
                            "admin", "admin", "tcp://ip:port");
            /* 获取连接 */
            Connection conn = connFactory.createConnection();
            conn.start();
            Session session = conn.createSession(
                    false, Session.AUTO_ACKNOWLEDGE);
            /* 创建基于Point-to-Point模型的消费者 */
            consumer = session.createConsumer(
                    session.createQueue("testQueue"));
            /* 设置消息监听 */
            consumer.setMessageListener(this);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    @Override
    public void onMessage(Message msg) {
        try {
            /* 消费消息 */
            logger.info(((TextMessage) msg).getText());
        } catch (JMSException e) {
            logger.error("消息消费失败",e);
```

```
        }
    }
}
```

消费者代码和生产者代码非常相似，区别是消费者需要实现 MessageListener 接口，并重写 onMessage()方法实现消息监听。关于 ActiveMQ 的更多使用方式本书不再一一进行讲解，大家可以参考其他的文献或书籍。

2.4.3 使用阿里开源的 RocketMQ 实现互联网场景下的流量消峰

笔者曾为大家介绍了 Apache 开源的轻量级消息中间件 ActiveMQ。由于 ActiveMQ 在部署和使用上非常简单，因此在一些用户规模较小的场景下使用 ActiveMQ 会非常实惠，但是它却并不适用于互联网，毕竟 ActiveMQ 这类消息中间件的设计初衷就是为企业级应用而服务。

相信大家都知道，任何一款消息中间件的基础功能都是实现异步调用和系统之间的解耦，但是在互联网场景下，大型网站需要面对的主要问题是高并发和海量数据。通过集群技术可以很好地提升系统的并行处理能力，落地服务化架构可以实现分而治之的管理，能够有效避免牵一发而动全身的风险，但是遵循 JMS 规范的消息中间件似乎并不重视分布式应用，并且通常对分布式特性的支持相对较弱。除此之外，在某些特殊场景下，我们还需要考虑顺序消息、事务消息、高吞吐量、高可用性及扩展性等问题，这些似乎都不是 ActiveMQ 的强项。因此笔者建议大家使用为互联网场景量身定制的分布式消息中间件产品来实现系统之间的解耦、异步调用及大流量消峰等操作，比如 Apache 开源的 Kafka、阿里开源的 RocketMQ 等产品，当然本书的主角是 RocketMQ。

RocketMQ 是阿里开源的一款分布式消息中间件，前身为 MetaQ，但是在 3.0 版

本之后，阿里正式将其更名为 RocketMQ。它具备以下六个特点：

- 支持顺序消息；
- 支持事务消息；
- 支持集群与广播模式；
- 亿级消息堆积能力；
- 完善的分布式特性；
- 支持 Push 与 Pull 两种消息订阅模式。

在历年的“双 11”大促活动中，RocketMQ 都承担着阿里生产系统 100%的消息流转，在核心交易链路有着稳定和出色的表现，是承载交易峰值（2016 年为 17 万笔/秒）的核心基础产品之一，并且阿里也计划将 RocketMQ 捐赠给 Apache 社区，这对于推动国产技术迈向国际化舞台来说无疑是一件值得庆幸的事情。RocketMQ 的项目地址为 https://github.com/alibaba/RocketMQ。

如图 2-9 所示，RocketMQ 由 NameServer、Broker、Producer 及 Consumer 四部分构成，每个部分都可以采用集群部署，在最大程度上保证了消息服务整体的高可用性。

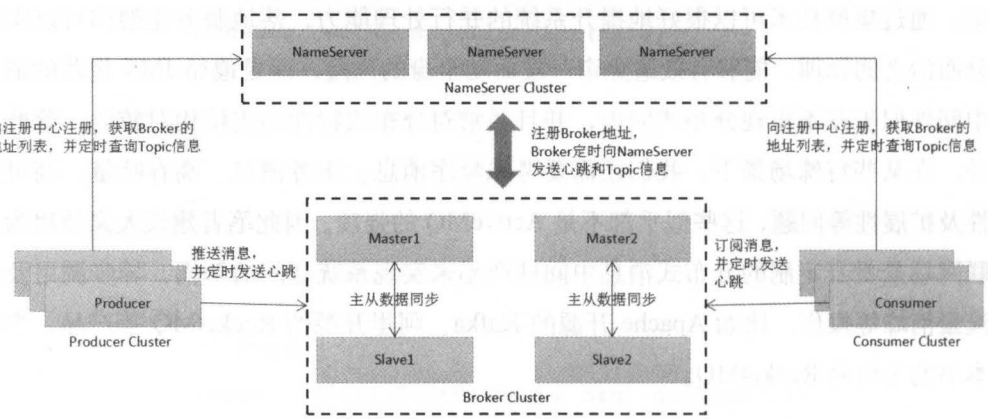


图 2-9 RocketMQ 的网络架构

简单来说，NameServer 其实就是一个注册中心，Broker、Producer 及 Consumer 在启动时都会向其进行注册，它的主要功能是负责客户端的寻址操作。NameServer 集群中的各个节点之间都是无状态的，由于 NameServer 并不会存在特别频繁的读/写操作，几乎没有负载压力，因此在生产环境中部署 NameServer 集群时并不需要添加太多的节点。

Broker 为消息服务端，提供消息的管理、存储及分发等功能。一个 Topic 会被均匀分布到集群环境中的所有 Broker 节点上，在 RocketMQ 中实际负责消息存储的是 Queue，它包含在 Topic 内部。换句话说，Queue 是消息存储的最小逻辑单元。如果生产环境中某个 Topic 的消息量特别大，那么应该为 Broker 集群添加更多的节点来降低单个 Broker 节点的负载压力。当每个 Broker 节点启动时会和 NameServer 集群中的所有节点建立长连接，并定时轮询发送心跳和 Topic 信息，由 NameServer 负责定时检查当前的存活连接，如果在指定的时间内没有接收到心跳，那么这个 Broker 节点的会话连接将会被 NameServer 主动关闭。

Producer 为消息生产者，用于向 Broker 推送消息；而 Consumer 为消息消费者，负责消费 Broker 中的消息。单个 Producer 和 Consumer 在启动时会和集群环境中的某一个 NameServer 节点建立长连接，定时轮询 Topic 信息，如果当前所连接的 NameServer 节点宕机，则自动重连到其他集群节点上。除此之外，Producer 和 Consumer 还会定时向 Broker 发送心跳，由 Broker 负责定时检查当前的存活连接，如果在指定时间内没有接收到心跳，那么客户端的会话连接将被 Broker 主动关闭。

对于 Broker 的部署，RocketMQ 提供以下四种部署形式：

- 单 Master 模式；
- 多 Master 集群模式；
- 多 Master/Slave 异步复制模式；
- 多 Master/Slave 同步双写模式。

单 Master 的部署形式是最简单的，但也是风险最大的，因为一旦 Master 宕机，那么整个消息服务将不可用。因此在生产环境中，笔者建议至少应该采用多 Master 集群模式进行部署，这样即便集群环境中的某一个 Broker 节点宕机，Consumer 和 Producer 还可以继续访问其他节点，并且采用多 Master 集群模式性能也是最好的，配置也不复杂。唯一的缺点就是集群环境中某一个 Broker 节点宕机期间，未被消费的消息在该节点还未恢复之前 Consumer 并不能够进行消费，因此如果希望 Broker 节点宕机后 Consumer 还能够继续消费该节点上未被消费的消息，那么只能够采用多 Master/Slave 集群模式进行部署。

在多 Master/Slave 集群模式下，RocketMQ 提供主从之间的同步双写和异步复制两种方式。如果采用同步双写模式来同步数据，那么就类似于 MySQL 数据库的半同步复制（Semi-synchronous Replication）功能，同一份数据只有在主从都成功写入后，才会返回给 Producer，这样即便 Master 宕机，Consumer 仍然可以从 Slave 上订阅还未被消费的消息，并且主从之间的消息是无延迟的，但性能相对较低。如果采用异步复制，由于主从之间的数据同步采用异步操作，对性能的影响自然就较小，几乎接近于多 Master 集群模式的性能，但会导致主从之间出现一定数据不一致的窗口期，但通常消息的延迟都是在毫秒级别。在此需要注意，目前开源版本的 RocketMQ 并不提供 Master 宕机后 Slave 自动切换为 Master 的功能，因此究竟应该使用哪一种部署模式，还需要根据实际的应用场景而定。

接下来笔者就为大家演示应该如何部署 Broker，本书仅以多 Master 集群模式为例，关于 Broker 的其他部署模式，本书不再一一进行讲解，大家可以参考其他的文献或者书籍。

本书所使用的 RocketMQ 版本为 3.2.6。当大家成功下载好 RocketMQ 后，可以通过命令“tar -zxvf”对其进行解压，然后编辑文件/etc/profile 配置 RocketMQ 的环境变量，如下所示：

```
export RMQ_HOME=/usr/local/alibaba-rocketmq
export PATH=$RMQ_HOME/bin:$PATH
```

当 RocketMQ 的环境变量配置完成后，通过命令“source /etc/profile”让其立即生效。在启动 Broker 之前，要先启动 NameServer，为了演示方便，笔者仅启动一个 NameServer 节点，如下所示：

```
#启动 NameServer
nohup sh mqnamesrv &
```

当成功启动 NameServer 节点后，再启动多个 Broker（Master）节点，如下所示：

```
#启动 Master1
nohup sh mqbroker -n ip:9876 -c$RMQ_HOME/conf/2m-noslave/broker-a.properties &
#启动 Master2
nohup sh mqbroker -n ip:9876 -c$RMQ_HOME/conf/2m-noslave/broker-b.properties &
```

在此需要注意，如果是在同一台物理机器上部署多 Master 集群模式，那么必须为不同的 Broker 节点指定不同的端口号，否则将触发以下异常信息：

```
java.net.BindException: Address already in use
```

编辑 $RMQ_HOME/conf/2m-noslave/ 目录下的 broker-a.properties 和 broker-b.properties 两个文件，修改 Broker 的端口，如下所示：

```
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerId=0
deleteWhen=04
```

```
fileReservedTime=48
brokerRole=ASYNC_MASTER
flushDiskType=ASYNC_FLUSH
#伪集群部署时，需要为每一个 Broker 节点设置不同的端口号
listenPort=10913
```

当成功启动 NameServer 节点和 Broker 节点后，笔者再为大家演示应该如何使用 RocketMQ 的 Producer 和 Consumer 实现消息推送和消息订阅。开发人员可以通过 Maven 依赖的方式下载 RocketMQ 构件，如下所示：

```
<dependency>
    <groupId>com.alibaba.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>3.2.6</version>
</dependency>
<dependency>
    <groupId>com.alibaba.rocketmq</groupId>
    <artifactId>rocketmq-common</artifactId>
    <version>3.2.6</version>
</dependency>
<dependency>
    <groupId>com.alibaba.rocketmq</groupId>
    <artifactId>rocketmq-filtersrv</artifactId>
    <version>3.2.6</version>
</dependency>
```

成功下载好运行 RocketMQ 所需的相关构件后，应编写消息生产者，如下所示：

```
/**
 * 基于 RocketMQ 的消息生产者
 *
 * @author gaoxianglong
```

```
 */
public class ProducerTest {
    private static DefaultMQProducer producer;
    private static Logger logger= LoggerFactory.
            getLogger(ProducerTest.class);
    public @BeforeClass static void init() {
        producer = new DefaultMQProducer("testProducterGroup");
        producer.setNamesrvAddr("ip:port");
        try {
            producer.start();
        } catch (MQClientException e) {
            e.printStackTrace();
        }
    }
    public @Test void sentMsg() {
        try {
            /* 向消息队列写入消息 */
            SendResult result = producer.send(new Message(
                    "testTopic", "Hello RocketMQ".getBytes()));
            logger.info("msgId-->" + result.getMsgId() + "\t"
                    + "result-->" + result.getSendStatus());
        } catch (Exception e) {
            logger.error("消息推送失败", e);
        }
    }
}
```

在上述程序中，我们先创建了一个 DefaultMQProducer 的实例，代表 RocketMQ 的消息生产者，设置 producerGroup 和 NameServer 地址后调用其 start()方法即可启动 Producer 的连接。在此需要注意，在集群模式下，由于 Topic 会被均匀分布到集群环境中的所有 Broker 节点上，那么 Producer 在启动时，会和该 Producer 关联的所有

Broker 节点建立长连接。

DefaultMQProducer 提供 send()方法用于向消息队列中写入消息，其构造函数中参数 Message 为 RocketMQ 提供的消息对象。Message 构造函数的第 1 个参数为 TopicName，第 2 个参数为消息主体（需要显式转换为字节数组在网络中进行传递），当一切准备就绪后，调用 send()方法即可成功将消息写入到指定的 Topic 中。

接下来我们来看看如何编写消息消费者代码，如下所示：

```
/**
 * 基于 RocketMQ 的消息消费者
 *
 * @author gaoxianglong
 */
public class ConsumerTest {
    /* 使用 PUSH 消费方式 */
    private static DefaultMQPushConsumer pushConsumer;
    private static Logger logger = LoggerFactory.getLogger(
            ConsumerTest.class);
    public @BeforeClass static void init() {
        pushConsumer = new DefaultMQPushConsumer("testConsumerGroup");
        pushConsumer.setNamesrvAddr("ip:port");
        /* 设置 MQ 的工作线程数 */
        pushConsumer.setConsumeThreadMax(100);
        pushConsumer.setConsumeThreadMin(100);
        /* 采用集群模式 */
        pushConsumer.setMessageModel(MessageModel.CLUSTERING);
    }
    public @Test void getMsg() {
        try {
```

```
            pushConsumer.subscribe("testTopic", null);
            /* 指定 Consumer 从哪里开始消费 */
            pushConsumer.setConsumeFromWhere(ConsumeFromWhere.
                    CONSUME_FROM_FIRST_OFFSET);
            /* 订阅指定 Topic 中的消息 */
            pushConsumer.registerMessageListener(new
                    MessageListenerConcurrently() {
                public ConsumeConcurrentlyStatus consumeMessage(
                        List<MessageExt> msgs,
                        ConsumeConcurrentlyContext Context) {
                    if (!msgs.isEmpty()) {
                        for (MessageExt msg : msgs) {
                            logger.info("msgId-->" + msg.getMsgId() +
                                    "\tbody-->"+new String(
                                            msg.getBody()));
                        }
                    }
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                }
            });
            pushConsumer.start();
        } catch (MQClientException e) {
            e.printStackTrace();
        }
        try {
            /* 按任意键退出 */
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

在上述程序中，笔者指定了 Consumer 采用的消息订阅模式为 PUSH，尽管 RocketMQ 还支持 PULL 模式，但无论开发人员在程序中使用哪种消息订阅模式，RocketMQ 在底层仍然使用 PULL 模式负责消息订阅。

创建 DefaultMQPushConsumer 对象实例后，意味着创建了一个 RocketMQ 中的消息消费者，和创建消息生产者类似，我们仍然需要指定 NameServer 的地址。对于那些消息体量较大的场景，笔者建议为 Consumer 设置合理的工作线程数（默认最小线程数为 20，最大线程数为 64），如果线程数设置得过多或过少，必然会适得其反。DefaultMQPushConsumer 的 setMessageModel()方法用于设置 Consumer 采用哪种消息模型，RocketMQ 默认支持集群和广播两种消息模型，如果采用的是集群模型（点对点模型），那么同一个消息只会由 Consumer 集群中的某一个节点进行消费；广播模型则恰恰相反，所有订阅了目标 Topic 的 Consumer 节点都可以进行消费。

DefaultMQPushConsumer 提供的 setConsumeFromWhere() 方法用于指定 Consumer 从哪里开始消费，本书采用的是 CONSUME_FROM_FIRST_OFFSET，也就是说，一个新的订阅组第一次启动从队列的最前位置开始消费，再启动从上次消费的位置开始消费。当然还可以指定其他方式，如下所示：

```
/**
 * 一个新的订阅组第一次启动从队列的最后位置开始消费，
 * 再启动从上次消费的位置开始消费
 */
CONSUME_FROM_LAST_OFFSET,
/**
 * 一个新的订阅组第一次启动从队列的最前位置开始消费，
 * 再启动从上次消费的位置开始消费
 */
CONSUME_FROM_FIRST_OFFSET,
/**
```

```
 * 一个新的订阅组第一次启动从指定时间点开始消费，
 * 再启动从上次消费的进度开始消费，
 * 时间点设置参见 DefaultMQPushConsumer.consumeTimestamp 参数
 */
CONSUME_FROM_TIMESTAMP,
```

通过 DefaultMQPushConsumer 的 registerMessageListener()方法可以让 Consumer 从指定的 Topic 中订阅消息，该方法参数 MessageListenerConcurrently 接口的 consumeMessage()方法需要开发人员重写，这样一旦有消息写入，Consumer 便可进行消费。consumeMessage()方法的返回值为枚举类型 Consume ConcurrentlyStatus，代表了消息的消费状态，如果消息消费成功则可以返回 CONSUME_SUCCESS，反之返回 RECONSUME_LATER 后允许重复进行消费，在最大程度上确保消息能够消费成功，并且在某些特殊场景下也便于让业务实现 Failover 操作。

最终仍然需要调用 DefaultMQPushConsumer 的 start()方法来启动 Consumer 连接，和启动 Producer 一样，启动 Consumer 时也会和该 Consumer 关联的所有 Broker 节点建立长连接。

RocketMQ 的部署和使用方法其实还是比较简单的，但 RocketMQ 默认情况下并没有像 ActiveMQ 那样为开发人员提供管理控制台界面，但这并不意味着无法对 Topic、Cluster 及 Broker 等信息进行管理。接下来笔者为大家演示如何使用 RocketMQ 提供的 mqadmin 工具来实现一些基本的命令管理，如下所示：

创建一个指定的 Topic：

```
sh mqadmin updateTopic -n ip:port -c DefaultCluster -t topicName
```

删除一个指定的 Topic：

```
sh mqadmin deleteTopic -n ip:port -c DefaultCluster -t topicName
```

查看 Broker 中所有的 Topic 信息：

```
sh mqadmin topicList -n ip:port
```

查看指定 Topic 的统计信息：

```
sh mqadmin topicStatus -n ip:port -t topicName
```

查看所有的消费组 Group：

```
sh mqadmin consumerProgress -n ip:port
```

查看指定消费组下所有 Topic 的消息堆积情况：

```
sh mqadmin consumerProgress -n ip:port -g consumerGroupName
```

2.4.4 基于 MQ 方案实现流量消峰的一些典型案例

笔者在前面为大家详细讲解了关于 MQ 的一些基础概念、Apache 开源的企业级消息中间件 ActiveMQ，以及专门为互联网场景量身定制的阿里开源消息中间件 RocketMQ 的具体部署和使用方式。下面就和笔者一起回顾一下，在实际开发过程中使用 MQ 的两个主要目的。首先，通过异步调用可以很好地解决分布式环境下系统之间的耦合问题，减少系统中一些非必需的依赖调用，降低复杂度和维护成本，从而保证进程功能的单一性；其次，在互联网场景下，大型网站主要需要面对的问题是高并发和海量数据，那么为了避免流量过大对系统产生较大冲击，引发系统出现雪崩现象，利用 MQ 来实现流量消峰可以让流量可控，使之有条不紊地对系统进行访问操作，在最大程度上保护系统的稳定运行。

在很多技术难题的解决方案中大家都会看见 MQ 的身影，如今 MQ 在大型网站

的架构设计中占据着非常重要的地位。一般而言，在使用 MQ 进行流量消峰的场景中，更多还是体现在控制并发写流量降低后端存储系统的负载压力上。比如数据库或者分布式缓存系统，如果并发写的流量过大，往往会导致吞吐量降低，RT 线性上升，容量容易被瞬间撑爆导致资源连接耗尽等悲剧发生，因此需要一种能够体现在技术上的消峰手段，对并发写流量进行排队处理。再回顾一下漏桶算法，其本质是我们并不需要关心流量的入口有多大，只需要关心出口流量有多大即可，那么我们可以先将消息写入到消息队列中，由订阅了目标 Topic 的消费者来负责实际的并发写操作。假设 RocketMQ 的 Consumer 使用的是 PUSH 模式来订阅消息，开发人员可以将 MQ 的工作线程数设置在一个比较合理的范围内，比如最大工作线程数被设置为 100，那么在同一时间并发写的最大流量也就始终被控制在 100 之内。

笔者也为大家讲解了如何控制针对数据库的并发写流量，这便是大促场景下使用 MQ 进行流量消峰的典型案例，当然类似的案例还有很多，由于篇幅有限，笔者就不一一进行讲解，只为大家分享以下两个案例：

- 前端埋点数据上报消峰案例；
- 分布式调用跟踪系统的埋点数据上报消峰案例。

上述两个案例针对的都是埋点数据上报，只是前者需要由开发人员手动在 APP 中进行埋点，后者是嵌入在服务框架中（本书以 Dubbo 框架为例）进行埋点。笔者先将埋点数据进行上报，希望利用大数据平台对用户的行为进行分析。在笔者的生产环境中，专门搭建了处理埋点数据的 Web 服务，并由该服务将埋点数据落盘到大数据平台。假设 Web 服务直接将数据写入到 HBase，那么在大促场景下 HBase 的负载压力将变得非常大，并且 Web 服务的响应时间也会线性上升，因此笔者将埋点数据异步写入到 MQ 中，由订阅了目标 Topic 的消费者负责 HBase 的实际写入，如图 2-10 所示。

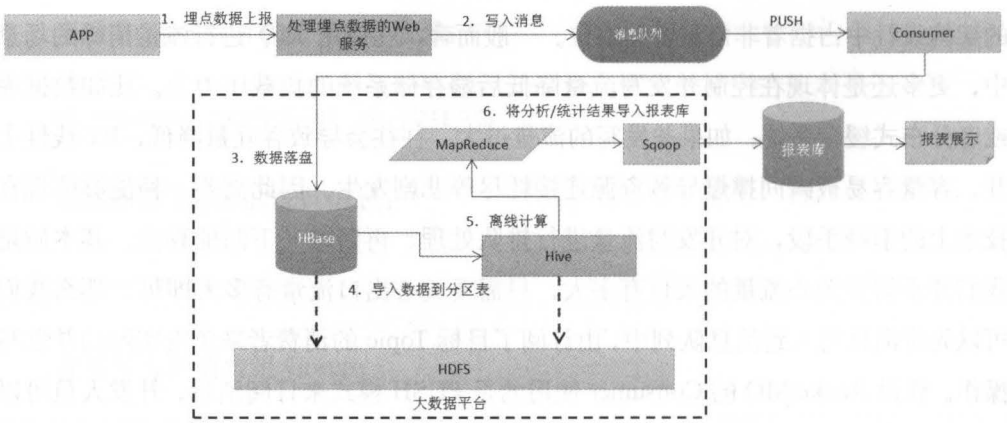


图 2-10 前端埋点数据上报与离线分析流程

为了降低 HBase 的负载压力，以及加速 Web 服务的响应时间，通过 MQ 进行流量消峰，可以很好地控制并发写的流量，并且对于实时分析统计需求而言，Spark 默认提供了对 Kafka 的读/写操作支持，因此将埋点数据写入到 MQ 就显得顺理成章了。

在第 1 章中，笔者为大家介绍了如何基于 Dubbo 实现一个分布式调用跟踪系统，由于跟踪系统在运行时会产生大量的数据，因此架构师需要解决两个棘手问题。首先，如果数据存储在 MySQL 数据库中，当单表数据量过大时必然会影响检索效率，因此要么将数据落盘到 NoSQL 数据库中，要么定时清理 MySQL 中的历史数据；其次，无论将数据落盘到哪里，如果并发写的流量过大，存储系统的负载压力都不小，因此通过 MQ 来实现流量消峰将是一个非常不错的解决方案。

在此需要注意，并不是任何场景都适合使用 MQ 来进行系统之间的解耦和流量消峰等操作，对于需要同步等待调用结果的业务场景而言，如果贸然异步化必然会对业务流程造成严重影响，甚至还会影响用户体验。

2.5 本章小结

笔者从业务层面和技术层面两个维度为大家详细讲解了应该如何对流量实施管制，从而避免在大促场景下因峰值流量过大对系统造成较大冲击，引发系统出现雪崩现象。笔者先为大家讲解了目前市面上一些常见的限流算法，比如令牌桶算法、漏桶算法及计数器算法等，并演示了如何使用 Nginx 的限流功能来实现接入层限流，以及使用计数器算法在限时抢购场景下控制单机并发写流量。除了可以运用技术手段实施流量管制，也可以在业务上做调整，换一种思路或玩法，采用基于时间分片的消峰方案也可以有效对流量实施管制。最后，笔者为大家讲解了如何使用 MQ 进行异步调用、实现系统之间的解耦，并为大家演示了阿里开源的消息中间件 RocketMQ 的具体部署和使用方式，还分享了笔者在实际工作中基于 MQ 实现流量消峰的一些典型案例。

第3章 分布式配置管理服务案例

相信大家对配置信息都不会感到陌生，在实际的开发过程中，无论是访问数据库、分布式缓存系统、消息队列，还是通过 Dubbo 框架实现 RPC 调用，都需要提前配置好目标 URL、账号/密码等信息，因此这类信息也被称为配置信息。在大部分情况下，我们都会选择将相关配置信息配置在配置文件中，当系统启动时，会从指定的目录下进行加载，通过获取配置文件中的配置信息项来完成环境的初始化工作。除此之外，我们在使用电脑进行办公、娱乐时，也会高频率地与配置信息打交道，比如，通过操作系统的控制面板来设置显示器的分辨率、鼠标的双击速度，以及区域和语言设置等，这些都属于配置信息，所以如果你告诉我你从未接触过配置信息，那么我一定会摇摇头对你说这不可能。

本章为大家介绍两种最常见的配置形式，一种基于本地配置形式，一种则适用

于大规模分布式场景的集中式资源配置形式，本章重点围绕后者展开，为大家详细讲解如何基于 ZooKeeper 实现一个分布式配置管理平台，并演示淘宝的 Diamond 系统和百度的 Disconf 系统的具体使用方式。

3.1 本地配置

将配置信息配置在本地，通常的做法有如下两种：

- 将配置信息耦合在业务代码中；
- 将配置信息配置在配置文件中。

3.1.1 将配置信息耦合在业务代码中

在业务代码中耦合配置信息绝对是大部分新手常干的事情，笔者刚参加工作的时候，就因为这样做不知道被领导批评了多少次，随着工作时间的增加，以及经验的积累（其实是踩坑和填坑经历得多了），慢慢也改掉了这种看似随意方便但却不利于项目后期维护的做法。接下来我们就先看看，将配置信息耦合在业务代码中，究竟会带来哪些不便。

数据库连接是一种非常昂贵且数量有限的底层资源，因此在对数据库进行读/写操作时，开发人员在程序中应该尽量使用数据库连接池（DB ConnectionPool）技术，以实现资源复用和限制单机能够申请到的最大并发连接数。本书以阿里开源的 Druid 连接池为例，将数据源信息耦合在业务代码中，如下所示：

```
private static DruidDataSource dataSource;
/**
 *  初始化数据源
 *
```

```
     * @author gaoxianglong
     */
    public @BeforeClass static void init() {
        try {
            /* 配置数据源信息 */
            dataSource = new DruidDataSource();
            dataSource.setUsername("root");
            dataSource.setPassword("123456");
            dataSource.setUrl("jdbc:mysql://ip:port/db");
            dataSource.setInitialSize(10);
            dataSource.setMinIdle(10);
            dataSource.setMaxActive(20);
            dataSource.setPoolPreparedStatements(false);
            dataSource.setMaxOpenPreparedStatements(-1);
            dataSource.setTestOnBorrow(false);
            dataSource.setTestOnReturn(false);
            dataSource.setTestWhileIdle(true);
            dataSource.setFilters("mergeStat,log4j,config");
            dataSource.setConnectionProperties("config.decrypt=false");
            dataSource.setUseGlobalDataSourceStat(false);
            dataSource.setTimeBetweenLogStatsMillis(300000);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    /**
     * 数据库CRUD操作
     *
     * @author gaoxianglong
     */
    public @Test void testCURD() {
        try {
            Connection conn = dataSource.getConnection();
```

```
            /* TODO CRUD */
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

一旦将数据源信息耦合在业务代码中，在后续的部署工作上必然会存在一些麻烦。由于系统可能会被部署在不同的环境中（如开发环境、测试环境、生产环境等），但不同环境之间存在的差异性（如各个环境的 URL 不同、账号/密码不同、单机所允许申请的最大连接数不同等），会使开发人员每次都只能通过修改业务代码的方式进行适应，如图 3-1 所示。

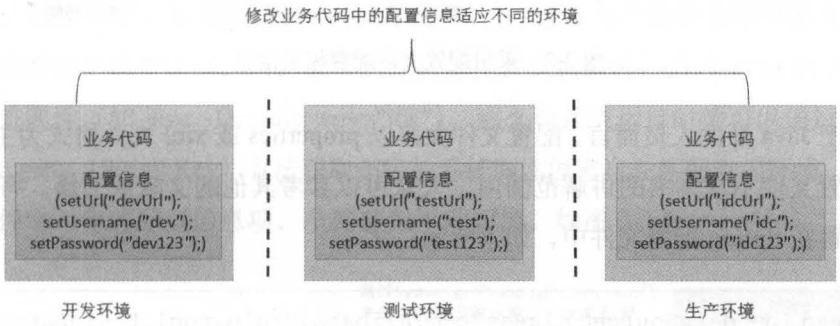


图 3-1 将配置信息耦合在业务代码中

为了提高工作效率，在一些比较简单的单元测试场景中，我们可以将配置信息耦合在测试代码中。除此之外，笔者不建议将任何可变的配置信息耦合在业务代码中，而是建议采用配置文件，因为即便后续配置信息需要进行调整，开发人员也不必通过修改业务代码来达到目的。

3.1.2 将配置信息配置在配置文件中

采用本地配置文件，我们可以很好地将可变的配置信息与业务代码进行解耦。

假设之前数据源中配置的 MaxActive 为 50，现需要调整为 100 时，开发人员只需要修改配置文件即可，相对于将配置信息耦合在业务代码中，采用这种方式则显得更轻量，如图 3-2 所示。

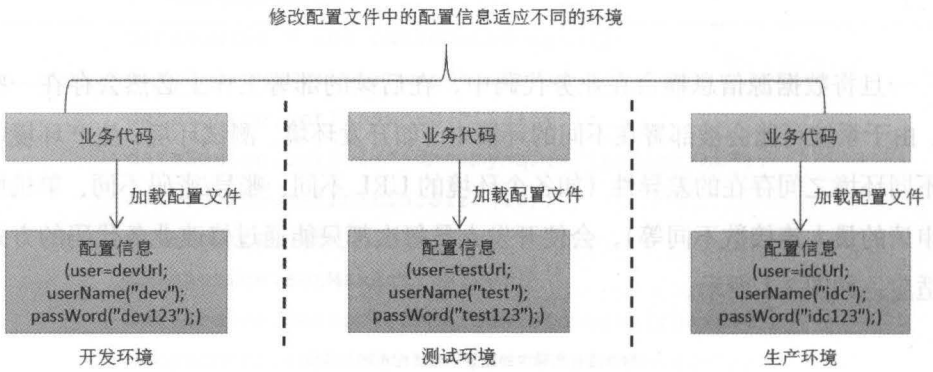


图 3-2 采用配置文件配置相关信息

对于 Java 开发人员而言，配置文件大多以 properties 或 xml 文件格式为主，如何解析配置文件不在本书的讲解范围内，大家可以参考其他的文献或书籍。将数据源信息配置在 Spring 配置文件中，如下所示：

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
    init-method="init" destroy-method="close">
    <property name="username" value="root" />
    <property name="password" value="pwd" />
    <property name="url" value="jdbc:mysql://ip:port/db" />
    <property name="initialSize" value="10" />
    <property name="minIdle" value="10" />
    <property name="maxActive" value="20" />
    <property name="poolPreparedStatements" value="false" />
    <property name="maxOpenPreparedStatements" value="-1" />
    <property name="testOnBorrow" value="false" />
    <property name="testOnReturn" value="false" />
```

```
        <property name="testWhileIdle" value="true" />
        <property name="filters" value="mergeStat,log4j,config" />
        <property name="connectionProperties" value="config.decrypt=
false" />
        <property name="useGlobalDataSourceStat" value="false" />
        <property name="timeBetweenLogStatsMillis" value="300000" />
    </bean>
```

笔者建议大家在配置文件中预先定义好不同环境所需的配置信息项，并由系统在运行时自动进行匹配和加载。如果当前处于测试环境，那么就加载测试环境的配置信息项；如果当前处于生产环境，那么就加载生产环境的配置信息项。这样一来，从版本提测到最终测试通过，运维人员便可以直接将测试通过后的版本发布到生产环境中。试想一下，如果系统运行在不同的环境中还需要手动切换不同的配置信息项，这就增加了维护成本，因此通常的做法是在系统启动时，通过 JVM 的启动参数设置系统属性（如 java -Dconfig_env="idc"），那么当系统运行时便可以通过 System 的 getProperty（String Key）方法获取指定的系统属性值来自动匹配当前环境，并加载配置文件中对应的配置信息项，从而避免手动切换，如图 3-3 所示。

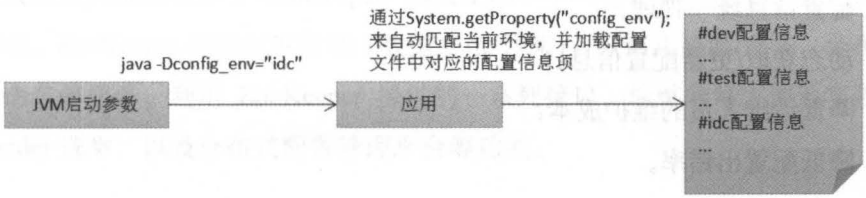


图 3-3 根据不同的环境加载对应的配置信息

大家思考一下，如果是处于分布式环境中，由于对业务进行了垂直划分，并且系统往往都是采用集群部署的，那么集群环境中的每一个节点都持有同一份配置文件，一旦配置信息发生改变，就意味着集群环境中的所有配置文件都需要做出相应的调整，而随着系统拆分的粒度越来越细，维护成本将会大大提升，并且配置出错

的可能性也随之增加，因此需要一种集中式资源配置的形式，以让所有的集群节点共享同一份配置信息。除此之外，在某些特殊的业务场景下，我们希望配置信息是可以在运行时发生变更的，系统必须在不重启的情况下获取到这些变更后的数据信息，并及时作出相应的调整策略。

基于本地配置文件的做法显然不能够有效应对上述需求，因此过渡到分布式资源管理平台就显得顺理成章。

3.2 集中式资源配置需求

笔者在本章的前面几个小节中为大家介绍了关于本地配置的两种做法，尽管目前一些中小型互联网企业仍然将本地配置作为首选，但是当网站发展到一定规模时，继续采用本地配置暴露出的问题将越来越多，因此到了这个阶段，架构师们必然需要对系统现有的配置形式做出调整。在分布式场景下，使用集中式资源管理平台将带来以下四点好处：

- 配置信息统一管理；
- 动态获取/更新配置信息；
- 降低运维人员的维护成本；
- 降低配置出错率。

希望大家记住，在系统架构演变的过程中，任何一个细节都不容被忽视，否则就是在为将来埋坑。简单来说，分布式配置管理服务的本质就是典型的发布/订阅（Publish/Subscribe）模式，获取配置信息的一方为订阅方，发布配置信息的一方为推送方，如图 3-4 所示。

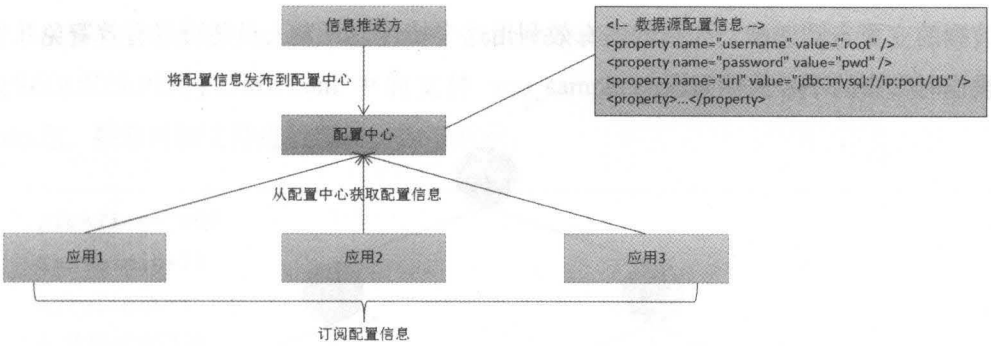


图 3-4 集中式资源配置

3.2.1 分布式一致性协调服务 ZooKeeper 简介

ZooKeeper 是 Apache 旗下的一款采用 Java 语言编写并开放源代码的分布式一致性协调服务，它是基于 Google Chubby 的一个开源实现。熟悉 Hadoop 平台的开发人员应该不会对其感到陌生，在 Hadoop 集群中由 ZooKeeper 负责 NameNode 的管理、HBase 的 Master 选举等任务，在第 1 章中笔者为大家演示的 Dubbo 框架也是使用 ZooKeeper 来实现注册中心进行服务的动态注册与发现。在实际的开发过程中，大部分开发人员仅间接接触到了 ZooKeeper，那么使用 ZooKeeper 究竟能够做些什么呢？简单来说，ZooKeeper 提供配置管理（数据发布/订阅）、分布式协调/通知、分布式锁及统一命名等服务，利用 ZooKeeper 提供的一系列接口，能够非常方便地实现一致性、Leader 选举，以及分布式配置管理平台等功能。

如图 3-5 所示，在 ZooKeeper 的数据模型中，每个子节点称为 Znode，每个 Znode 都有一个全局唯一的路径作为标示，进行数据的读/写操作。

大家需要明确一点，Znode 的不同类型（持久节点和瞬时节点）决定了其不同的行为特性。如果需要实现 Leader 选举、分布式锁等场景时，那么瞬时节点无疑是最好的选择。除此之外，每一个 Znode 节点都维护着一个版本号，版本号会随着每次

数据的变更自动递增，如果能够有效利用这个特性，开发人员便可以有效避免并发操作时带来的不一致性问题。

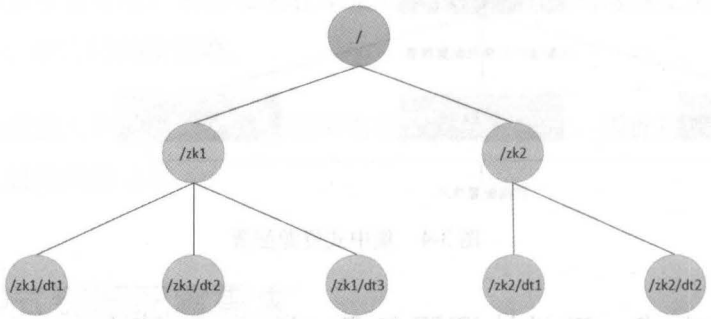


图 3-5 ZooKeeper 的数据模型

3.2.2 ZooKeeper 的下载与集群安装

“工欲善其事，必先利其器”是古训，接下来笔者将为大家介绍如何下载和安装 ZooKeeper。如果你熟知 ZooKeeper 的安装过程，那么笔者建议你直接跳过本小节，反之细心阅读之后再动手亲自操作一番方能加深印象。

本书使用的 ZooKeeper 版本为 3.4.6，笔者建议大家使用与此一致的版本，以避免一些因版本不一致引起的错误发生。在正式安装之前，大家还需要确保 JDK 等所需的相关环境已经准备就绪。由于 ZooKeeper 的安装过程非常简单，因此笔者直接演示集群模式的安装过程，关于单机模式大家可以参考其他的文献或者书籍。

使用命令“tar -zxvf”对已经下载好的 ZooKeeper 压缩包进行解压，然后编辑文件/etc/profile 配置 ZooKeeper 的环境变量，如下所示：

```
#ZooKeeper
ZOOKEEPER_HOME=/usr/local/ZooKeeper-3.4.6
PATH=$PATH:$ZOOKEEPER_HOME/bin
```

当环境变量配置完成后，通过命令“source /etc/profile”让其立即生效，再将目录$ZOOKEEPER_HOME/conf/中的文件 zoo_sample.cfg 重新复制一份并更名为 zoo.cfg，接着对该文件进行编辑，如下所示：

```
tickTime=2000
initLimit=10
syncLimit=5
# 数据存放目录
dataDir=$ZOOKEEPER_HOME/data
# 日志存放目录
dataLogDir=$ZOOKEEPER_HOME/datalog
# 客户端通信连接端口
clientPort=2181
# 最大客户端连接数
maxClientCnxns=50
# 集群节点相关信息
server.1=ip1:2888:3888
server.2=ip2:2888:3888
server.3=ip3:2888:3888
```

在上述配置文件中，“server.A=B:C:D”用于标识不同 ZooKeeper 节点的配置，其中 A 为节点编号，用于区分集群环境中的不同节点；B 为节点的 IP 地址；C 用于设置目标节点与集群环境中 Leader 的信息交换端口；D 用于设置选举端口，若集群环境中的 Leader 宕机后，节点之间需要通过此端口进行 Leader 的重新选举。

dataDir 和 dataLogDir 等目录需要大家手动在每个节点上进行创建。在 dataDir 目录中我们需要创建 1 个 myid 文件，该文件中的内容对应的就是当前节点的编号，比如节点 1，在 myid 文件中的内容就应该为 1（大小应该控制在 1～255）。最后通过命令“zkServer.sh start”依次启动集群环境中的每一个节点（命令“zkServer.sh stop”

为停止），当成功启动所有节点后，通过命令“zkServer.sh status”可以查看集群节点的当前状态。

在此需要注意，在节点启动过程中，ZooKeeper.out 将有拒绝连接等异常信息输出，这是因为集群环境中的某些节点还未完全启动，因此可以忽略掉这类异常信息。

3.2.3 ZooKeeper 的基本使用技巧

当大家成功下载并安装好 ZooKeeper 集群后，笔者就为大家演示一下 ZooKeeper 的一些基本使用技巧。在$ZOOKEEPER_HOME/bin/目录下有一个名为 zkCli.sh 的 Shell 脚本文件（Windows 平台下对应的为 zkCli.cmd 文件），该脚本文件作为 ZooKeeper 的客户端命令行工具，成功启动后我们将看到 ZooKeeper 输出的“Welcome to ZooKeeper!”等相关信息。

ZooKeeper 的基本操作非常简单，在客户端命令行工具中，我们通过如下命令即可实现对 ZooKeeper 的 CRUD 操作：

```
[zk: localhost:2181(CONNECTED) 7] create /zkTest testData
Created /zkTest
[zk: localhost:2181(CONNECTED) 8] get /zkTest
testData
cZxid = 0x170000000c
ctime = Sun Oct 02 13:52:47 CST 2016
mZxid = 0x170000000c
mtime = Sun Oct 02 13:52:47 CST 2016
pZxid = 0x170000000c
cversion = 0
dataVersion = 0
```

```
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 8
numChildren = 0
[zk: localhost:2181(CONNECTED) 9] set /zkTest testData2
cZxid = 0x170000000c
ctime = Sun Oct 02 13:52:47 CST 2016
mZxid = 0x170000000d
mtime = Sun Oct 02 13:53:02 CST 2016
pZxid = 0x170000000c
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 9
numChildren = 0
[zk: localhost:2181(CONNECTED) 10] delete /zkTest
```

在上述程序中，笔者先通过命令“create /zkTest testData”创建了一个新的 Znode 目录，以及与该节点对应的数据信息；然后通过命令“get /zkTest”验证了目标节点是否存在，并查询了与该节点对应的数据信息；命令“set /zkTest testdata2”则为修改命令，笔者在 3.2.1 节中曾经提及过，每一个 Znode 节点都维护着一个版本号，版本号会随着每次数据的变更自动递增，通过观察我们可以发现，当对节点“/zkTest”中的数据进行更新时，其版本号（dataVersion）从原来的数值“0”变为了“1”；最后，执行“delete”命令后将把之前创建的 Znode 目录删除，再试图通过“get”命令进行查询时，ZooKeeper 将会提示“Node does not exist”。

3.2.4 基于 ZooKeeper 实现分布式配置管理平台方案

使用 ZooKeeper 提供的配置管理服务，可以使开发人员非常容易地实现一套分

布式配置管理平台。简单来说，我们将配置信息发布到 Znode 目录上后，由客户端负责信息订阅，一旦配置信息发生变更，所有 Watch 目标 Znode 的 ZooKeeper 客户端都会感知到，那么当重新获取配置信息后，我们便可以在程序中执行一些自定义的逻辑处理（比如开关控制、动态更新系统的配置信息等操作），如图 3-6 所示。

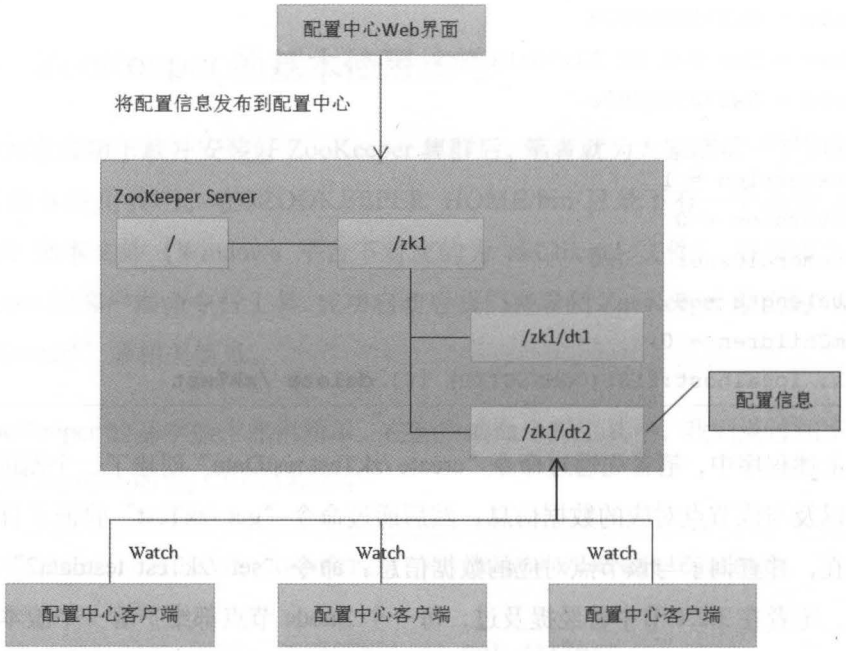


图 3-6 基于 ZooKeeper 的配置管理服务

基于 ZooKeeper 实现一个分布式配置管理平台，还需要考虑并实现客户端的以下三点功能：

- 信息配置管理界面；
- 订阅 Znode 中的配置信息；
- 容灾机制。

接下来笔者就为大家演示如何通过使用 ZooKeeper 提供的客户端 API 来实现配置信息的订阅功能。开发人员可以通过 Maven 依赖的方式下载 ZooKeeper 的相关构件，如下所示：

```
<dependency>
    <groupId>org.apache.ZooKeeper</groupId>
    <artifactId>ZooKeeper</artifactId>
    <version>3.4.6</version>
</dependency>
```

成功下载好运行 ZooKeeper 所需的相关构件后，我们来看看在程序中客户端应该如何与 ZooKeeper 服务端建立会话连接，如下所示：

```
private void connectionZK() {
    final CountDownLatch countDownLatch = new CountDownLatch(1);
    try {
        client = new ZooKeeper("127.0.0.1:2181", 5000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                final KeeperState STATE = event.getState();
                switch (STATE) {
                case SyncConnected:
                    countDownLatch.countDown();
                    logger.info("connection ZooKeeper success");
                    break;
                case Disconnected:
                    logger.warn("ZooKeeper connection is disconnected");
                    break;
                case Expired:
                    logger.error("ZooKeeper session expired");
                    break;
```

```
                    default:
                        break;
                    }
                }
            });
            countDownLatch.await();
        } catch (IOException | InterruptedException e) {
            logger.error("connecton ZooKeeper fail", e);
        }
    }
```

在上述程序中，笔者通过 new 关键字创建了一个 ZooKeeper 客户端的对象实例，ZooKeeper 构造函数的第 1 个参数 connectString 用于设置 ZooKeeper 服务端的通信地址和端口(如果是集群环境,则可以通过符号“,”进行分隔)；第 2 个参数 sessionTimeout 用于设置等待客户端通信的最长时间，笔者建议设置为 5～10 秒；最后的 Watcher 参数设置接收 ZooKeeper 的会话事件，由于 ZooKeeper 的 Watcher 被定义为接口，因此开发人员需要对其进行实现和重写其 process()方法。在实际开发过程中，我们通过 Watcher 不仅可以监听客户端与 ZooKeeper 服务端之间的会话状态,还能够监听 Znode 的节点变更。

在重写 Watcher 接口的 process()方法时，可以通过事件状态来判断客户端是否与 ZooKeeper 服务端成功建立会话连接，如果所返回的事件状态为 SyncConnected，则意味着会话建立成功，才允许程序执行下一步操作。假设已经成功建立好的会话突然被断开后，如果客户端可以连接到集群环境中的其他节点上时，将会触发 Disconnected 事件，并将这个会话迁移到另外的节点上，如图 3-7 所示。这里存在一种特殊情况，如果会话被断开后，在 sessionTimeout 所指定的时间内客户端都没能连接到另外的节点上时，这个会话便会失效，就算客户端重新尝试连接到 ZooKeeper 服务端后，使用之前无效的会话进行连接必然会触发 Expired 事件。

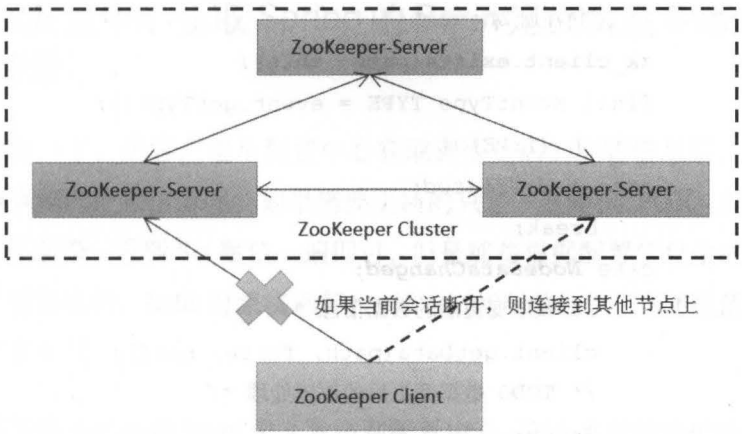


图 3-7 ZooKeeper 的会话迁移

当大家明确如何使用客户端 API 与 ZooKeeper 服务端建立会话连接后，笔者再为大家演示如何 Watch 目标 Znode 上的数据变更，如下所示：

```
/**
 * 实现 Watcher 监听目标 Znode 上的数据变更
 *
 * @author gaoxianglong
 */
public class ConfigWatcher implements Watcher {
    private ZooKeeper client;
    private String path;
    public ConfigWatcher(ZooKeeper client, String path) {
        this.client = client;
        This.path = path;
    }
    @Override
    public void process(WatchedEvent event) {
        try {
```

```
                /* 重新注册 Znode */
                zk_client.exists(path, this);
                final EventType TYPE = event.getType();
                switch (TYPE) {
                case NodeCreated:
                    break;
                case NodeDataChanged:
                    /* 获取变更后的数据信息 */
                    client.getData(path, false, null);
                    /* TODO 数据变更后的逻辑处理 */
                    break;
                case NodeChildrenChanged:
                    break;
                case NodeDeleted:
                default:
                    break;
                }
            } catch (KeeperException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

在上述程序中，笔者实现了 ZooKeeper 提供的 Watcher 接口，并重写了其 process() 方法来监听目标 Znode 上的节点变更，一旦发生变更，便会回调 process()方法。和监听会话状态不同，监听 Znode 目录的节点变更，获取的为事件类型而非事件状态，如果返回的事件类型为 NodeDataChanged，则意味着目标 Znode 上的数据已经发生了变更，当获取到变更后的配置信息后，就可以使系统在不重启的情况下实现配置信息的动态更新。

3.2.5 从配置中心获取 Spring 的 Bean 定义实现 Bean 动态注册

大家思考一下，当客户端从配置中心获取到变更后的配置信息后，我们应该如何动态更新系统的配置信息呢？如果需要更新的只是一些单例 POJO 对象中的数据信息或许并不复杂，重新 Set 赋值一遍即可，但是那些根据配置信息生成的对象实例（如 JDBC 对象实例、Jedis 对象实例等），却不能够简单地重新 Set 赋值，而是需要生成新的对象实例。

假设我们将 Spring 的 Bean 定义发布到配置中心，那么本地持有的内容则只是一个指向目标 Znode 目录上的“引用”。当对 IOC 容器进行初始化时，如果成功与 ZooKeeper 服务端建立会话连接，并获取到相关 Bean 定义，便可以在完成 Bean 定义的解析后将相关 Bean 动态注册到 IOC 容器中，如图 3-8 所示。当客户端 Watch 到配置信息发生变更后，应该重复注册这个动作。Bean 动态注册的伪代码如下所示：

```
    public static void registerBeans(String value) {
        final String tmpdir = "临时文件的全限定名";
        /* 生成用于存放从配置中心获取到的配置信息的临时 XML 文件 */
        try (BufferedWriter out = new BufferedWriter(new FileWriter
(tmpdir))) {
            if (null != value) {
                out.write(value);
                out.flush();
                FileSystemResource resource = new FileSystemResource
(tmpdir);
                ConfigurableApplicationContext cfgContext =
                    (ConfigurableApplicationContext) 当前上下文中的 IOC 容器;
                DefaultListableBeanFactory beanfactory =
                    (DefaultListableBeanFactory) cfgContext.getBeanFactory();
                /*
```

```
                * 加载和解析 XML 文件中的 Bean 定义，实现 Bean 的动态注册，
                * 不需要手动销毁 IOC 容器中相同 beanName 的 Bean 实例，
                * loadBeanDefinitions()方法会调用其他方法自动进行销毁
                */
                new XmlBeanDefinitionReader(beanfactory).
                    loadBeanDefinitions(resource);
                String[] beanNames = beanfactory.getBeanDefinitionNames();
                for (String beanName : beanNames) {
                    /* 实例化所有未实例化的 Bean */
                    beanfactory.getBean(beanName);
                }
            }
        } catch (Exception e) {
            logger.error("Bean 动态注册失败", e);
        } finally {
            /* 删除临时 XML 文件 */
        }
    }
```

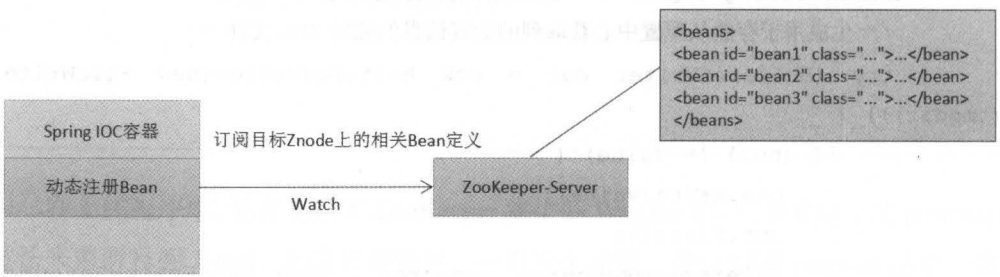


图 3-8 Bean 的动态注册

在上述程序中，registerBeans()的方法参数 value 为从配置中心获取到的 Spring 的相关 Bean 定义信息，在对 Bean 进行动态注册之前，还需要进行一系列的前期准备工作。

首先需要将 Spring 的相关 Bean 定义信息输出到本地一个临时文件中，然后通过 Spring 提供的 FileSystemResource 进行读取。接着将 ApplicationContext 的实例强制向下转换为 ConfigurableApplicationContext，这么做的目的主要是希望通过调用 getBeanFactory()方法来获取 DefaultListableBeanFactory 实例。如何获取到当前上下文的 ApplicationContext 实例是一个问题，值得庆幸的是 Spring 为开发人员提供了 ApplicationContextAware 接口，重写该接口的 setApplicationContext()方法后，在 IOC 容器初始化时我们便可以顺利获取到 ApplicationContext 实例。重要的是，实例化 XmlBeanDefinitionReader 后，可以通过调用它的 loadBeanDefinitions()方法从之前输出到本地的 XML 文件中加载相关的 Bean 定义，并由该方法调用 Spring 的其他方法完成如下操作：

- 解析 Bean 定义；
- 销毁当前 IOC 容器中与 Bean 定义中相同 beanName 的 Bean 实例；
- Bean 动态注册。

在此需要注意，如果 Bean 定义中配置了数据源信息，一定要将属性“destroy-method”设置为“close”，否则重新注册 Bean 后，之前持有的底层连接资源将无法得到释放。由于 BeanFactory 采用的是懒加载模式，也就是说，只有真正使用到目标 Bean 的时候才会对其进行实例化，因此我们可以通过手动调用其 getBean()方法来实例化 IOC 容器中未实例化的 Bean 对象。

由于 IOC 容器的移除操作不等价于 JVM 的内存回收操作，因此不能再通过 Spring 的注解装配得到相关的 Bean 实例，从而避免在配置变更后使用的还是之前的对象引用。

3.2.6 容灾方案

由于 ZooKeeper 底层采用的是冗余存储机制，因此从理论上来说，ZooKeeper

的集群节点数量越多，容灾性就越好，在网络不可用的情况下，开发人员需要自行实现客户端的容灾机制。

在正常情况下，当客户端成功连接到 ZooKeeper 服务端并获取到目标 Znode 节点上的配置信息后，可以固化到客户端本地的容灾目录下。当配置信息发生变更后，除了需要更新系统配置，还需要覆盖本地容灾目录下的缓存文件。这样，如果是因为网络问题导致客户端无法正常与 ZooKeeper 建立会话连接，客户端也能够支持离线启动，直接读取缓存在客户端本地的缓存数据继续运行，如图 3-9 所示。如果希望拥有更完善的容灾机制，笔者建议大家参考淘宝 Diamond 或者百度 Disconf 的相关容灾方案。

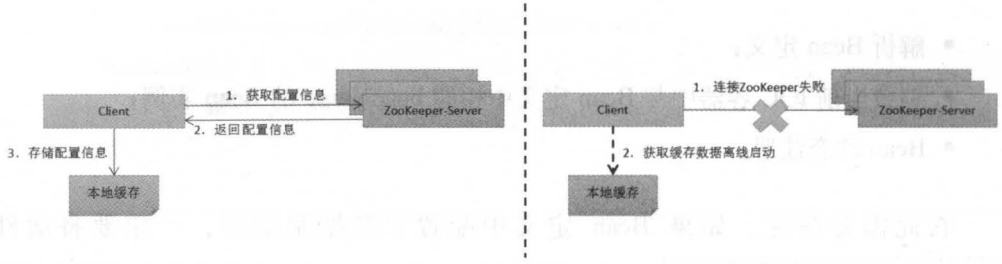


图 3-9 客户端容灾

3.2.7 使用淘宝 Diamond 实现分布式配置管理服务

相信淘宝 Diamond 的大名早已是众人皆知，如雷贯耳了。简单来说，Diamond 作为淘宝系内部广泛使用的一套分布式配置管理平台，其拥有配置/部署简单、高可用及易用性强等特点，使得早期在内部的推广工作进展得非常顺利，目前淘宝内部绝大多数系统的配置信息几乎都是通过 Diamond 进行统一管理的。大家可以从 GitHub 上下载 Diamond 的相关源代码进行阅读和扩展，Diamond（分支版本）的项目地址为 https://github.com/takeseem/diamond。

Diamond 的整体架构主要由 Diamond-Client 和 Diamond-Server 两部分构成。Diamond-Server 部署在服务端，主要用于配置信息的发布和管理；Diamond-Client 部署在客户端，用于配置信息的订阅，如图 3-10 所示。Diamond 采用的是简单的 Key-Value 结构，Diamond-Client 通过指定的 Key（dataId 和 group）订阅信息，那些未被订阅的信息将不会被推送。

图 3-10 Diamond 的整体架构

Diamond 之所以大受欢迎，与其完善的容灾机制密不可分。配置在 Diamond 中的配置信息最终会被固化到数据库、服务端本地缓存、客户端本地缓存中，甚至开发人员还能够通过手工的方式来干预 Diamond 的容灾目录，如图 3-11 所示。假如数据库 Master 机器宕机后，服务可以快速切换到 Slave 机器上；当数据库 Master/Slave 都不可用时，Diamond 还可以通过缓存在服务端本地的缓存数据继续对外提供订阅服务；当所有 Diamond 相关的服务都不可用时，客户端也能够支持离线启动，直接读取缓存在客户端本地的缓存数据继续运行；甚至在最极端的情况下，客户端的本地缓存数据被删除，开发人员也能够通过复制“镜像”到容灾目录下继续使用。Diamond 通过自身完善的容灾机制能够很好地保证配置管理服务整体的高可用性。

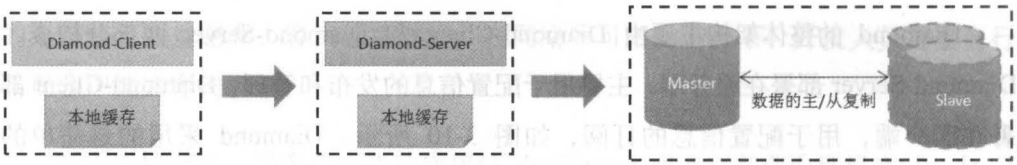


图 3-11 Diamond 容灾机制

接下来笔者就为大家演示 Diamond 的部署和使用。成功下载好运行 Diamond 所需的相关构件后，使用 Diamond 提供的 DDL 语句在数据库中创建所需的数据表，如下所示：

```
CREATE TABLE config_info (
id BIGINT(64) UNSIGNED NOT NULL AUTO_INCREMENT,
data_id VARCHAR(64) NOT NULL,
group_id VARCHAR(64) NOT NULL,
content LONGTEXT NOT NULL,
md5 VARCHAR(32)NOT NULL,
src_ip VARCHAR(20) default NULL,
src_user VARCHAR(20) default NULL,
gmt_create DATETIME NOT NULL,
gmt_modified DATETIME NOT NULL,
PRIMARY KEY (id),
UNIQUE KEY uk_config_datagroup (data_id,group_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_bin;
```

编辑 diamond-server 目录下的 jdbc.properties 文件，成功配置好数据源信息后，通过命令“mvn clean package -Dmaven.test.skip”对 Diamond-Server 进行打包，打包成功后将在 diamond-server/target/目录下生成 diamond-server.war，再将 war 包复制到 Tomcat 的/webapp/目录下并启动 Tomcat。在浏览器中输入“http://host:port/diamond-server”后即可成功进入到 Diamond-Server 的登录界面（初始账号/密码为 abc/123，大家可以编辑 diamond-server 目录下的 user.properties 文件创建指定的用户），

成功登录后即可跳转到 Diamond 的配置信息管理界面进行配置信息的发布和管理，如图 3-12 所示。

图 3-12 Diamond 的配置信息管理界面

dataId 和 group 共同构成客户端订阅数据时使用的 Key。接下来我们就来看看如何从 Diamond-Server 中订阅配置信息，在此之前要修改 diamond-util 目录中 com.taobao.diamond.common 包下的 Constants 类，该类定义了 Diamond 的一些默认常量，需要将常量 HTTP_URI_FILE、CONFIG_HTTP_URI_FILE 的值进行调整，如下所示：

```
/* 获取数据的 URI 地址，如果不带 ip，那么轮换使用 ServerAddress 中的地址请求 */
public static String HTTP_URI_FILE = "/diamond-server/config.co";
/* 获取 ServerAddress 的配置 uri */
public static String CONFIG_HTTP_URI_FILE = "/url";
```

使用 diamond-client 订阅配置信息，如下所示：

```
public @Test void testUseDiamond() {
    DefaultDiamondManager manager = new DefaultDiamondManager(
            "DEFAULT_GROUP", "testId", new ManagerListener() {
        @Override
```

```
            public Executor getExecutor() {
                return null;
            }
            @Override
            public void receiveConfigInfo(String configInfo) {
                /* 数据变更后的逻辑处理 */
            }
        });
        List<String> list = new ArrayList<String>();
        list.add("127.0.0.1");
        DiamondConfigure cfg = manager.getDiamondConfigure();
        /* 设置当前支持的所有的 Diamond-Server 域名列表 */
        cfg.setDomainNameList(list);
        /* 设置 Diamond-Server 的端口号 */
        cfg.setPort(8080);
        /* 设置客户端的轮询时间间隔，单位为“秒” */
        cfg.setPollingIntervalTime(10);
        /**
         * 同步获取一份有效的配置信息，按照本地文件→diamond 服务器→
         * 上一次正确配置的 snapshot 的优先顺序获取，如果这些途径都无效，
         * 则返回 null
         */
    manager.getAvailableConfigureInfomation(2000);
    }
```

在上述程序中，笔者首先创建了 DefaultDiamondManager 的实例，其构造函数的第 1 个参数为 group，第 2 个参数为 dataId，通过 group 和 dataId 组成的唯一 Key 能够让我们从 Diamond-Server 中订阅到指定的配置信息；而第 3 个参数 ManagerListener 为 Diamond 提供的数据监听器，如果需要监听数据变更，就需要重写其 receiveConfigInfo()方法。Diamond 的监听机制采用的是客户端默认每 15 秒主动

向 Diamond-Server 发起 HTTP 请求，检查配置信息是否发生变更，一旦发生变更，便会回调 receiveConfigInfo()方法。

通过 DefaultDiamondManager 的 getDiamondConfigure()方法获取到 DiamondConfigure 实例后，可设置 Diamond-Server 域名列表、端口号及客户端的轮询时间间隔等信息。在此需要注意，如果采用集群模式部署 Diamond-Server，那么域名列表地址指向的应该是 Nginx，并且需要编辑 diamond-server 目录下的 node.properties 文件添加集群环境中的所有节点，这样，当某个节点的配置信息发生变更后，DiamondServer 的通知服务会通知其他节点更新本地的配置信息，默认情况下，每 10 分钟 DiamondServer 才会 Dump 一次配置信息，我们可以通过编辑 diamond-server 目录下的 system.properties 文件来调整 Dump 配置信息的时间间隔，如下所示：

```
#Dump 配置信息的间隔时间，单位为“秒”
dump_config_interval=10
```

调用 DefaultDiamondManager 的 getAvailableConfigureInfomation()方法即可获取到配置在 DiamondServer 中的配置信息，参数 timeout 用于设置超时时间，单位为“毫秒”。

3.2.8 Diamond 与 ZooKeeper 的细节差异

尽管 Diamond 和 ZooKeeper 在配置管理服务上有着异曲同工之处，但两者在细节实现上却存在一些差异，如下所示：

- 数据存储方案不同；
- 监听数据变更的机制不同；
- 容灾机制不同；
- Znode 不适合存储大数据。

Diamond 主要通过 MySQL 数据库来管理和存储数据信息，其数据结构包括 DataID、Group 及 Content。而 ZooKeeper 采用的是类似于 UNIX 的文件系统目录，每个 Znode 都可以存储数据信息，并且除瞬时节点外，每个 Znode 目录下还可以包含子节点。

关于监听数据信息是否发生变更的问题，Diamond 采用的做法是客户端默认每 15 秒主动轮询检查。而 ZooKeeper 客户端则利用长连接对指定的 Znode 目录进行 Watch，一旦配置信息发生变更，实现 Watcher 接口的 process()方法将会感知到相应的事件。就实时性而言，ZooKeeper 更胜一筹。

笔者在 3.2.7 节中曾经介绍过 Diamond 完善的容灾机制，那么 ZooKeeper 又是如何保证数据容灾的呢？由于 ZooKeeper 底层采用的是冗余存储机制，因此从理论上来说，ZooKeeper 的集群节点数量越多，容灾性就越好，但在网络不可用的情况下，开发人员需要自行实现基于客户端的容灾机制。

为了提升系统的 QPS，ZooKeeper 放弃了从磁盘中读取需要的数据信息的方式，而是将所有数据都全量缓存在内存中，在内存中进行查找。但由于 Znode 并不支持数据的追加操作（全部都是替换操作），因此当数据量过大时，除了会导致读/写效率降低，还会加速耗尽 ZooKeeper 的可用内存资源，因此默认每个 Znode 所能够存储的数据容量为 1MB（可以通过修改环境变量“jute.maxbuffer”进行调整）。相比之下，Diamond 则没有这个限制。

3.2.9 使用百度 Disconf 实现分布式配置管理服务

除了可以使用淘宝的 Diamond 实现分布式配置管理服务，在实际的开发过程中，还可以选择其他优秀的分布式配置管理平台，比如基于 ZooKeeper 实现的百度开源 Disconf 系统。相对于 Diamond 来说，Disconf 更为年轻和活跃，但客观来说，两者

都是非常优秀的分布式配置管理平台，并不存在谁能够取代谁的问题，究竟应该使用哪一个平台，则要根据实际的业务场景而定。Disconf 的项目地址为 https://github.com/knightliao/disconf。

和淘宝的 Diamond 类似，百度的 Disconf 系统的整体架构也由客户端和服务端两部分构成。如图 3-13 所示，Disconf 的功能模块主要包括四部分，Disconf-Core 是 Disconf 的核心模块，客户端和服务端都必须依赖它；Disconf-Client 为客户端模块；Disconf-Tools 为工具模块；Disconf-Web 模块为服务端模块。

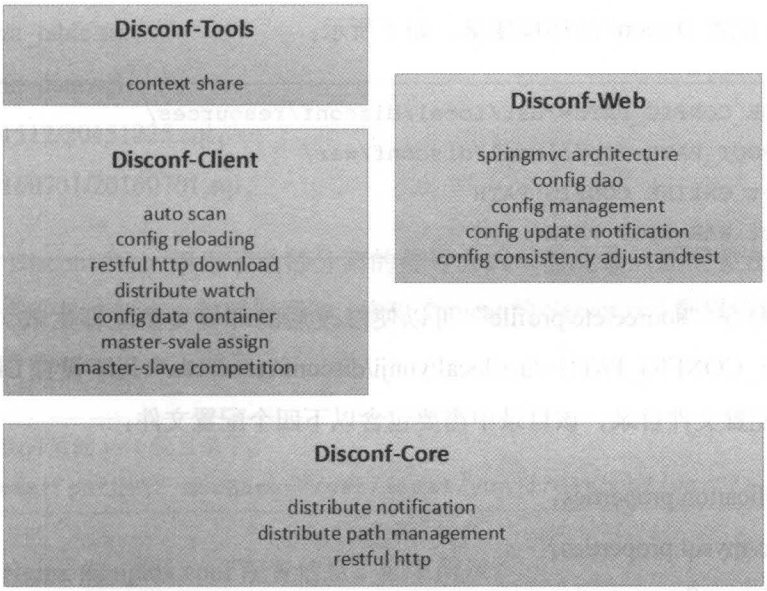


图 3-13 Disconf 的功能模块

接下来笔者就为大家演示如何部署 Disconf-Web 模块。由于 Disconf 的架构相对比较复杂，因此 Disconf 所依赖的外围系统自然也比较多。参考 Disconf 的官方文档，在部署 Disconf-Web 模块之前，必须确保系统中已经安装了以下依赖系统：

- MySQL(Ver 14.12 Distrib 5.0.45, for unknown-linux-gnu (x86_64) using EditLine wrapper)；
- Tomcat(apache-tomcat-7.0.50)；
- Nginx(nginx/1.5.3)；
- ZooKeeper(zookeeper-3.3.0)；
- Redis(2.4.5)。

当确认系统中上述环境都已经准备就绪后，我们便可以通过命令“git clone https://github.com/knightliao/disconf.git”下载 Disconf 的源码到本地，然后编辑文件/etc/profile 配置 Disconf 的环境变量，如下所示：

```
ONLINE_CONFIG_PATH=/usr/local/disconf/resources/
WAR_ROOT_PATH=/usr/local/disconf/war/
export ONLINE_CONFIG_PATH
export WAR_ROOT_PATH
```

通过命令“source/etc/profile”可以使修改后的环境变量立即生效。其中变量“ONLINE_CONFIG_PATH=/usr/local/yunji/disconf/resources/”用于设置 Disconf 需要使用到的配置文件目录，该目录中需要包含以下四个配置文件：

- application.properties；
- jdbc-mysql.properties；
- redis-config.properties；
- zoo.properties。

大家可以从目录/disconf/disconf-web/profile/rd/中复制上述四个配置文件，然后修改每一个配置文件中的数据源连接信息即可。变量“WAR_ROOT_PATH=/usr/local/yunji/disconf/war/”用于设置 Disconf-Web 模块源码编译后的 war 包部署地址。接下

来通过以下命令对 Disconf-Web 的源码执行编译：

```
cd disconf-web
sh deploy/deploy.sh
```

成功执行编译操作后，在变量 WAR_ROOT_PATH 所指定的目录下将生成编译后的 war 包内容。完成这些基础工作后，我们还需要在数据库中执行一些数据的初始化工作（如服务端用户的账号/密码数据等），关于数据的初始化脚本，则存放在目录 /disconf/disconf-web/sql 中，大家务必按照以下顺序逐一执行：

- 0-init_table.sql；
- 1-init_data.sql；
- /201512/20151225.sql；
- /20160701/20160701.sql。

由于 Disconf-Web 采用了动静分离的部署方式，因此最后还需要在 Nginx 和 Tomcat 中进行最终的 war 包部署操作。修改 Tomcat 的 Server.xml 配置信息，在 Host 节点下设定 Context，如下所示：

```
#指向编译后的 war 包目录下
<Context path="" docBase="/usr/local/yunji/disconf/war"/>
```

修改 Nginx 的 nginx.conf 配置信息，如下所示：

```
upstream disconf {
    server localhost:8088;
}
server {
    listen   80;
    server_name localhost;
```

```
        access_log /usr/local/disconf/tomcat/access.log;
        error_log /usr/local/disconf/tomcat/error.log;
        index  index.html index.htm;
        location / {
            root /home/gaoxianglong/war/html;
            if ($query_string) {
                expires max;
            }
        }
        location ~ ^/(api|export) {
            proxy_pass_header Server;
            proxy_set_header Host $http_host;
            proxy_redirect off;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Scheme $scheme;
            proxy_pass http://disconf;
        }
    }
```

在此需要注意，在上述配置文件中，笔者将 server_name 的值设置为“localhost”，这并不是必须的（也可以设置为域名等方式），但 Host 却必须与 application.properties 中 Key 为 domain 的值一致。

成功启动 Tomcat 和 Nginx 后，在浏览器中输入“http://host:port”即可成功进入 Disconf-Web 的登录界面（初始账号/密码为 admin/admin），成功登录后即可跳转到 Disconf 的配置信息管理界面，进行配置信息的发布和管理，如图 3-14 所示。简而言之，APP 代表某一个具体的项目，而不同的环境下可以包含当前 APP 的不同配置信息，并且配置信息的内容既可以是配置文件，也可以是配置项。

图 3-14 Disconf 的配置信息管理界面

服务端部署成功后，接着为大家演示 Disconf-Client 模块的使用。本书所使用的 Disconf 版本为 2.6.31，可以通过 Maven 依赖的方式下载 Disconf 的相关构件，如下所示：

```
<dependency>
    <groupId>com.baidu.disconf</groupId>
    <artifactId>disconf-client</artifactId>
    <version>2.6.31</version>
</dependency>
```

成功下载好运行 Disconf 所需的相关构件后，应该如何实现基于 Annotation 的方式订阅配置信息呢？首先在 Spring 的配置文件中添加 Disconf 的相关配置信息，如下所示：

```
    <context:component-scan base-package="com.test.use.disconf" />
    <aop:aspectj-autoproxy proxy-target-class="true" />
    <!-- 使用 disconf 必须添加以下配置 -->
    <bean id="disconfMgrBean" class="com.baidu.disconf.client.DisconfMgrBean"
        destroy-method="destroy">
        <property name="scanPackage" value="com.test.use.disconf" />
    </bean>
    <bean   id="disconfMgrBean2"   class="com.baidu.disconf.client.
DisconfMgrBeanSecond"
        init-method="init" destroy-method="destroy">
    </bean>
```

当程序启动时，默认会在 classpath 目录下加载一个名为 disconf.properties 的配置文件，在该文件中需要配置与 Disconf 服务端建立会话的连接信息，如下所示：

```
    # 是否使用远程配置文件，true（默认）则从远程获取配置，false 则直接获取本地配置
    enable.remote.conf=true
    # 配置服务器的 HOST，如果有多个则用逗号分隔
    conf_server_host=127.0.0.1:8088
    # 版本号
    version=1.0.0
    # APP
    app=app_test
    # 环境
    env=rd
    # debug
    debug=true
```

```
# 忽略哪些分布式配置，用逗号分隔
ignore=
# 获取远程配置，重试次数默认是 3 次
conf_server_url_retry_times=1
# 获取远程配置，重试时休眠时间默认是 5 秒
conf_server_url_retry_sleep_seconds=1
```

最后我们再定义一个 POJO，为这个 POJO 标记 @DisconfFile 注解（用于指定配置在 Disconf 服务端中的具体文件名），并在指定字段的 get 方法上标记 @DisconfFileItem 注解，用于接收从 Disconf 服务端中订阅的配置信息，如下所示：

```
@Service
@Scope("singleton")
@DisconfFile(filename = "userinfo.properties")
public class UserBean {
    private String account;
    private String pwd;
    @DisconfFileItem(name = "account", associateField = "account")
    public String getAccount() {
        return account;
    }
    public void setAccount(String account) {
        this.account = account;
    }
    @DisconfFileItem(name = "pwd", associateField = "pwd")
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

当 IOC 容器成功初始化时，Disconf 客户端便可以顺利地从 Disconf 服务端中获取到配置信息，并且一旦配置信息发生了变更，Disconf 客户端就会在 Watch 到变更后自动进行配置信息的更新操作。关于 Disconf 更多的使用方式，本书不再一一进行讲解，大家可以参考 Disconf 的官方文档：http://disconf.readthedocs.io。

3.3 本章小结

笔者为大家阐述了在实际开发过程中常见的两种配置形式，分别为本地配置形式和集中式资源配置形式，由于本地配置并不适用于分布式场景，因此本章重点围绕后者进行展开。笔者首先为大家详细介绍了分布式一致性协调服务 ZooKeeper 的一些基本使用技巧，并演示了如何基于 ZooKeeper 来实现一个分布式配置管理平台。最后还为大家详细演示了淘宝 Diamond 和百度 Disconf 的具体使用方式。由于基于 ZooKeeper 来实现分布式配置管理平台还需要由开发人员自行编写可视化的配置信息管理界面和实现客户端容灾机制，因此笔者建议大家在生产环境中，应该尽量使用现有的成熟的解决方案，毕竟有些轮子确实没有必要再重复去造。

除了淘宝 Diamond 和百度 Disconf，目前市面上还有一些非常优秀的分布式配置管理平台，奇虎 360 的 QConf 及 Spring-Cloud 都可以实现分布式配置管理服务。

4 第4章 大促场景下热点数据的读/写优化案例

在秒杀、限时抢购这种大促场景下，由于峰值流量较大，大量的并发读/写操作一定会导致后端的存储系统产生性能瓶颈。前面已经讲解过，提升单机处理能力最有效的办法就是采用集群技术对服务器进行扩容，只要系统能够具备良好的伸缩性，那么从理论上来说，其容量便可以是无限的。在此需要注意，大促场景下因热点数据导致的单点瓶颈已经不再是简单地通过横向扩容就能够顺利解决的，尽管对于读操作我们可以将热点数据缓存在分布式缓存中以达到提升系统 QPS 的目的，但是缓存系统的单点容量还是存在上限的，因此应对大促场景下的峰值流量仍显得杯水车薪。除此之外，由于热点数据的写操作无法直接在缓存中完成，那么这必然会引起大量的线程相互竞争 InnoDB 的行锁。并发越大时，等待的线程就越多，这会严重影响数据库的 TPS，导致 RT 线性上升，最终可能引发系统出现雪崩。

本章笔者会结合实际的工作经验，为大家重点讲解大促场景下热点数据读/写技术难题的一系列解决方案。

4.1 缓存技术简介

缓存（Cache）如今已不再是一门新鲜的技术，它早已是大部分开发人员工作中最频繁接触的技术种类之一。简而言之，缓存指的是将被频繁访问的热点数据存储在距离计算最近的地方，以方便系统快速做出响应，比如静态资源数据（包括图片、音频、视频、脚本文件及 HTML 网页等），我们可以缓存在 CDN（Content Delivery Network，内容分发网络）上。由于用户的请求并不是落到企业的数据中心，而是请求到离用户最近的 ISP（Internet Service Provider，互联网服务提供商）上，因此可以大幅提升系统整体的响应速度，如图 4-1 所示。

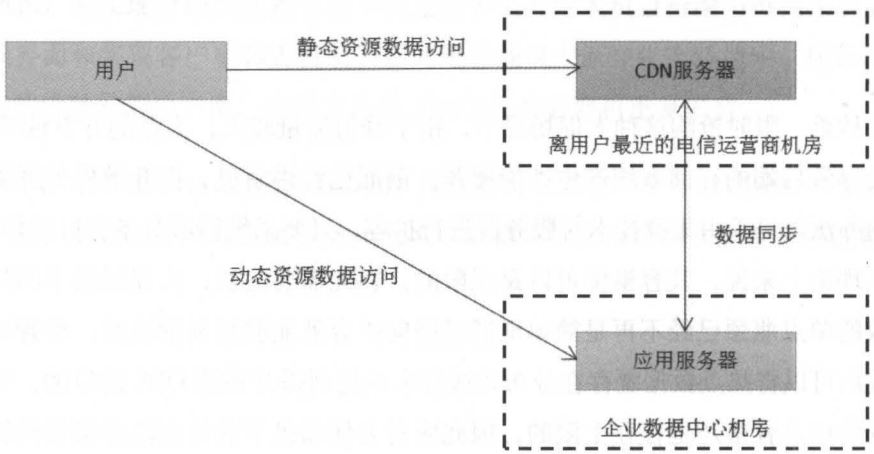


图 4-1 利用 CDN 缓存静态资源数据

反向代理服务器也可以缓存状态更新不频繁的静态资源数据，和 CDN 缓存不同，尽管反向代理服务器是部署在企业的数据中心机房内，但由于它挡在应用服务器的

前端，用户请求无须经过应用服务器就可以快速将结果返回给用户，因此也可以将数据缓存在反向代理服务器上，以达到加快系统响应速度的目的。

由于数据库的读/写能力远远比不上缓存的效率，因此除了可以使用 IDC、反向代理等手段缓存静态资源数据，业务系统从数据库等存储系统中获取的数据信息也可以进行缓存；这样当再次获取指定的数据时，可以优先从缓存命中，以减轻对数据库的负载，如图 4-2 所示。

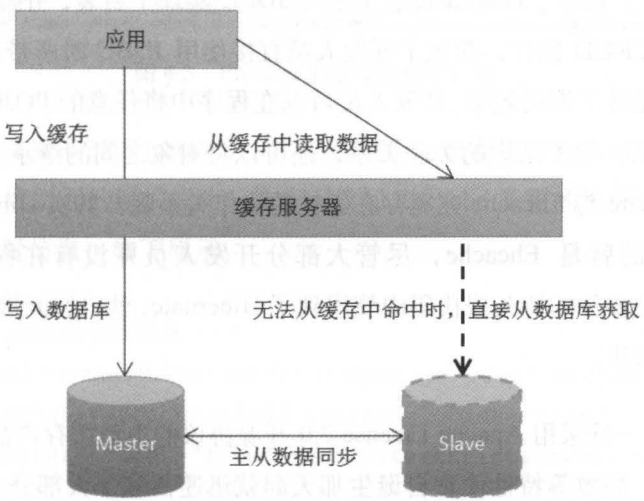


图 4-2 缓存数据库中的数据

开发人员在程序中经常使用的缓存产品大致可以分为以下两类：

- 本地缓存；
- 分布式缓存。

无论是本地缓存还是分布式缓存，目前市面上均提供了诸多成熟、易用、性能高效的缓存产品，比如 Ehcache、MemCache 及 Redis 等。总之无论使用哪一类缓存技术，总归是要解决以下两个问题：

- 缓解应用系统或关系型数据库的负载压力；
- 提升系统吞吐量。

4.1.1 使用 Ehcache 实现数据缓存

在持久层领域，大家对 Hibernate 应该不会感到陌生，蓦然回首，笔者在 7 年前就已经开始接触并在工作中使用这款开放源代码的 ORM（Object Relation Mapping，对象关系映射）产品了。Hibernate 在上层对 JDBC 实现了封装，并提供 HQL 语句实现对数据库的 CRUD 操作，简化了开发人员直接使用 JDBC 时所带来的诸多不便，在一定程度上提升了编码效率。开发人员可以在程序中将任意的 POJO 对象与数据库表建立映射关系，除了常规的关联关系，还可以将对象之间的继承关系映射到数据库中。那 Hibernate 与 Ehcache 究竟存在怎样的协作关系呢？其实 Hibernate 中的二级缓存默认使用的就是 Ehcache，尽管大部分开发人员并没有在程序中直接使用 Ehcache，但是只要你的持久层代码中使用到了 Hibernate，就已经间接使用了 Ehcache 来缓存查询结果集。

Ehcache 是一款采用 Apache License 2.0 开源协议的本地缓存产品，采用 Java 语言编写，简单、高效等特性使它自诞生那天起就迅速占领了大部分本地缓存市场。Ehcache 的整体功能架构如图 4-3 所示。如果脱离了 Hibernate，在程序中应该如何单独使用 Ehcache 来缓存热点数据呢？首先从 Maven 的中央仓库中下载运行 Ehcache 所需的相关构件，如下所示：

```
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.10.2</version>
</dependency>
```

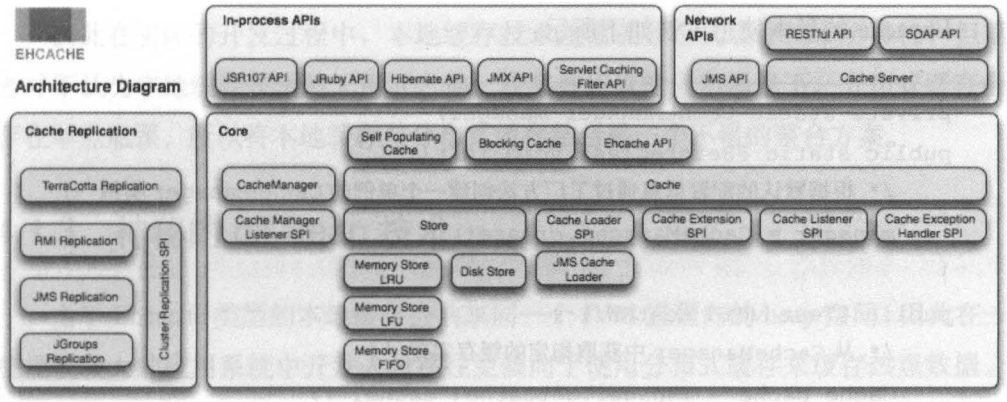


图 4-3 Ehcache 的整体功能架构图

本书以 2.10.2 版本的 Ehcache 为例，笔者建议大家使用与此一致的版本，以避免一些因版本不一致引起的错误发生。成功下载好 Ehcache 所需的相关构件后，再来配置 Ehcache 的配置信息，如下所示：

```
<!-- 指定 cache 的参数信息 -->
<cache name="cache1" maxElementsInMemory="1000" eternal="false"
       timeToLiveSeconds="60" overflowToDisk="true"
       diskPersistent="true" memoryStoreEvictionPolicy="LRU" />
</ehcache>
```

在上述配置信息中，属性“maxElementsInMemory”用于指定缓存的最大个数，假设值被设定为“1000”，就意味着全局最多能够缓存 1000 个数据；属性“eternal”指定了缓存是否永久有效，如果值被设定为“true”，那么 timeout 将不起作用；属性“timeToLiveSeconds”用于指定缓存数据的失效时间；属性“overflowToDisk”指定了当缓存数量超出“maxElementsInMemory”的阈值时，是否固化到磁盘；属性“diskPersistent”用于指定是否缓存 JVM 重启期数据，默认为“false”；属性“memoryStoreEvictionPolicy”用于指定缓存的回收策略，默认采用 LRU（最近最少使用），还可以设定为 FIFO（先进先出）或 LFU（较少使用）等策略。

Ehcache 的基本使用方法如下所示：

```
private static CacheManager manager;
public static @BeforeClass void init() {
    /* 根据默认的配置信息通过工厂方法创建一个单例的 CacheManager 实例 */
    manager = CacheManager.create();
}
public @Test void testRW() {
    /* 从 CacheManager 中获取指定的缓存实现 */
    Cache cache = manager.getCache("cache1");
    /* 向缓存中写入数据 */
    cache.put(new Element("key1", "value1"));
    /* 根据指定的 key 从缓存中获取数据 */
    cache.get("key1").getObjectValue();
}
```

当程序中使用 CacheManager.create()方法创建 CacheManager 实例时，如果没有显式指定配置文件的路径，那么 Ehcache 会从 classpath 下加载一个名为 ehcache.xml 的默认配置文件。

4.1.2 LocalCache 存在的弊端

尽管在程序中使用 Ehcache 能够在一定程度上缓解数据库的查询压力，提升系统整体的吞吐量；但是在互联网领域，应用系统的可用内存资源本身就非常紧张，而 Ehcache 的本质是同一个进程内的缓存技术，因此肯定会共享 JVM 有限的内存资源。假设缓存数据所占的内存比例较大，那么 JVM 极有可能在后续的操作中因腾不出足够多的 heap 空间用于为新对象分配内存空间，从而导致程序在运行的过程中抛出 java.lang.OutOfMemoryError 异常。其次，由于 GC 次数的增多或者暂停时间的延长，反而还会影响系统的吞吐量。

因此在实际的开发过程中，本地缓存技术逐渐被分布式缓存技术所替代；但笔者并不认为本地缓存技术就无用武之地，在某些特殊的应用场景下，分布式缓存会存在单点瓶颈，所以将本地缓存与分布式缓存结合是一个不错的整合方案。

4.1.3 神秘的 off-heap 技术

由于 Ehcache 类型的本地缓存会共享同一个 JVM 进程内的 heap 空间，因此在一些规模较大的应用系统中开发人员往往更倾向于使用分布式缓存来缓存热点数据。不过 Ehcache 在正式发布 3.x 里程碑版本后，开始提供 off-heap（堆外内存）特性。该特性可以避免缓存的热点数据占用 heap 空间中较多的内存资源，以降低 GC 次数或暂停时间，这对于 Ehcache 的拥护者和使用者来说确实是一件值得庆幸的事情。

笔者之所以说 off-heap 技术非常神秘，是因为这与 JVM 的底层机制密切相关。对于 Java 开发人员而言，Java 程序自始至终都只能运行在 JVM 内部，这与实际的物理宿主环境之间是相互“隔离”的；因此在默认情况下，为新对象分配内存空间都是在 heap 空间中进行，一般情况下，开发人员直接使用堆外内存的场景几乎是很少见的。而且得益于 JVM 的自动内存管理机制，Java 开发人员并不需要像编写 C/C++ 代码那样，在创建一个新对象时，还需要显式编写内存分配和内存回收等相关函数。总之，使用 Java 语言编写程序时，可以使我们从烦琐的劳动中解放出来，只关注自身业务即可。

在 Java 中，除了 heap 空间，几乎所有的内存空间都可以称为 off-heap。本书以实际的物理内存资源为例，为大家一层一层地揭开 off-heap 技术的神秘面纱。在程序中直接使用 off-heap 存储数据究竟会带来哪些好处呢，这或许是大家最关心的一个问题，如下所示：

- 减少 GC 次数或降低暂停时间；
- 可以扩展和使用更大的内存空间；

- 省去物理内存与 heap 之间的数据复制步骤。

默认情况下，Java 对象都是在 heap 空间中分配内存，但这也并非绝对，比如淘宝在 OpenJDK 的基础之上深度定制的 TaobaoVM 就使用 GCIH（GC invisible heap）技术实现了 off-heap。这样生命周期较长的 Java 对象就可以从 heap 中移至 heap 外，并且 GC 不能管理 GCIH 内部的 Java 对象，降低了 GC 的回收频率，提升了 GC 的回收效率。除此之外，随着逃逸分析技术的逐渐成熟，也可以让 Java 对象直接在栈上进行分配。但不可否认的是，heap 空间目前仍然是 GC 的重点回收区域，毕竟大部分对象还是在 heap 中进行分配的。

使用 off-heap 技术还可以在一定程度上提升系统的执行性能。假设操作的是物理内存，那么 JVM 的所有本机 I/O 操作都将尽可能直接在物理内存上进行。反之，JVM 会先从物理内存上进行 I/O 操作，然后再将数据复制到 heap 中（或者从 heap 中复制到物理内存）。

那究竟应该如何在 Java 程序中直接使用物理内存呢？Java 开源社区成熟的 NIO 网络通信框架 Netty、消息中间件 RocketMQ 等都使用了 ByteBuffer.allocateDirect() 方法来操作实际的物理内存资源，如下所示：

```
public @Test void testMemory() {
    final int _1G = 1024 * 1024 * 1024;
    final String value1 = "test_memory";
    log.info("before freeMemory:" + Runtime.getRuntime().freeMemory());
    /* 直接在物理内存中分配一块固定大小的直接字节缓冲区 */
    ByteBuffer buffer = ByteBuffer.allocateDirect(_1G);
    log.info("after freeMemory:" + Runtime.getRuntime().freeMemory());
    /* 写入数据 */
    buffer.put(value1.getBytes());
    /* 切换为读模式 */
    buffer.flip();
```

```
        log.info("从物理内存中读出数据-->");
        for (int i = 0; i < value1.length(); i++) {
            /* 不影响指针的偏移量 */
            log.info((char) buffer.get(i));
        }
        byte[] value2 = new byte[value1.length()];
        buffer.get(value2, 0, value2.length);
        log.info("从物理内存中读出数据-->" + new String(value2));
    }
```

在上述程序中，通过调用 ByteBuffer 接口的静态方法 allocateDirect(int capacity)可以创建出一个 DirectByteBuffer 类的实例，并根据参数“capacity”的值，直接在物理内存中分配一块固定大小的直接字节缓冲区。开发人员可以通过指定 JVM 的参数“-XX:MaxDirectMemorySize”来限制程序调用 allocateDirect(int capacity)方法时可申请的全局总容量大小，DirectByteBuffer 类中包含的协作类 Bits 的 totalCapacity 变量记录着这个值，一旦超出阈值，那么程序将在运行时抛出 java.lang.OutOfMemoryError 异常。当所申请的容量被批准时，DirectByteBuffer 将调用 sun.misc.Unsafe 类的一系列 native 方法执行内存分配操作。如果向直接字节缓冲区中写入的数据容量超出由参数“capacity”指定的阈值时，程序将在运行时抛出 java.nio.BufferOverflowException 异常。

尽管在程序中可以通过 ByteBuffer.allocateDirect()方法来操作实际的物理内存资源，但是这些被申请的堆外内存在什么情况下才会被释放呢？简单来说，DirectByteBuffer 对象仅仅只是一个引用，其背后关联着一大段堆外内存，由于 DirectByteBuffer 的对象实例仍然存储在 heap 空间内，因此可以将其称为“冰山对象”，如图 4-4 所示。当 DirectByteBuffer 对象被 GC 回收时，其背后的堆外内存也会被一并释放。

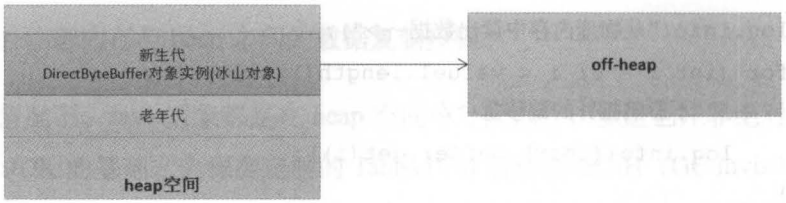


图 4-4 持有 off-heap 引用的“冰山对象”

在此需要注意，不是在任何场景下使用 off-heap 技术都会给程序的执行性能带来帮助，在某些情况下甚至还会适得其反，毕竟通过 ByteBuffer.allocateDirect()方法来申请物理内存资源所需的成本要高于直接在 heap 中的操作。

4.2 高性能分布式缓存 Redis 简介

本地缓存尽管可以提供快速的数据访问能力，但是局限也非常明显。对于那些需要考虑数据共享的应用场景本地缓存则显得无能为力，并且本地缓存的容量有限，无法实现横向扩容；因此对于那些业务发展较迅猛的互联网企业来说，从本地缓存架构演变到分布式缓存架构几乎是必经之路。

目前市面上流行的分布式缓存产品琳琅满目，开源的 Redis、Memcache 等几乎占据着分布式缓存市场的半壁江山，可谓是每家互联网企业的标配。当然这个世界上总会多多少少地存在一些“不按常理出牌”的家伙，他们总是希望在具体的业务场景下有更优的解决方案或性能更强悍的产品来服务业务，因此淘宝自主研发的 Tair、腾讯自主研发的 CKV 都是性能非常高效的分布式缓存产品。

本书以 Redis 为例，简单来说，Redis 是一个使用 ANSI C 语言编写的开源分布式 Key-Value 存储系统，支持多种数据类型（如 String、List、Set、zset、Hash 等），并提供丰富的客户端 API（如 Java、C/C++、PHP、Ruby 等）。对于 Redis 的使用。除了常规的单点，如果希望利用数据水平存储或横向扩容来提升 Redis 的整体容量，

那么可以采用一致性 Hash 算法将多个 Redis 节点构建成一组 Redis 集群。由于扩容可以是动态的，因此从理论上来说，Redis 集群后的容量是无限的。

4.2.1 使用 Jedis 客户端操作 Redis

本书使用的 Redis 的版本为 3.2.3，笔者建议大家使用与此一致的版本，以避免一些因版本不一致引起的错误发生。成功下载并安装好 Redis 后，我们还需要下载 Redis 的客户端 API 来实现缓存数据的 CRUD 操作，本书所使用的客户端 API 为 Jedis，如下所示：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

成功下载好运行 Jedis 所需的相关构件后，笔者为大家演示如何在程序中使用 Jedis 客户端操作 Redis，如下所示：

```
private static JedisPoolConfig cfg;
private static JedisPool jedisPool;
private static String HOST = "127.0.0.1";
private static int PORT = 6379;
public static @BeforeClass void init() {
    /* 配置连接池信息 */
    cfg = new JedisPoolConfig();
    cfg.setMinIdle(10);
    cfg.setMaxIdle(20);
    cfg.setMaxTotal(50);
    cfg.setMaxWaitMillis(3000);
    cfg.setTestOnBorrow(true);
```

```
        cfg.setTestOnReturn(true);
        jedisPool = new JedisPool(cfg, HOST, PORT);
    }
    public @Test void testJedis() {
        Jedis jedis = jedisPool.getResource();
        jedis.set("key1", "value1");
        Log.info(jedis.get("key1"));
    }
```

Redis 的整体性能是非常高效的，无论是使用 Redis 官方提供的 benchmark 工具，还是使用笔者自行编写的压测代码对 Redis 进行吞吐量测试，结果都非常令人满意。从压测结果来看，Redis 单点 TPS 可以达到 8 万/秒，QPS 则可以达到惊人的 10 万/秒。

4.2.2 使用 Redis 集群实现数据水平化存储

在 3.x 版本之前，Redis 并没有提供 Cluster 功能，如果我们想对 Redis 进行集群，则只能自行采取一致性 Hash 算法，如图 4-5 所示。

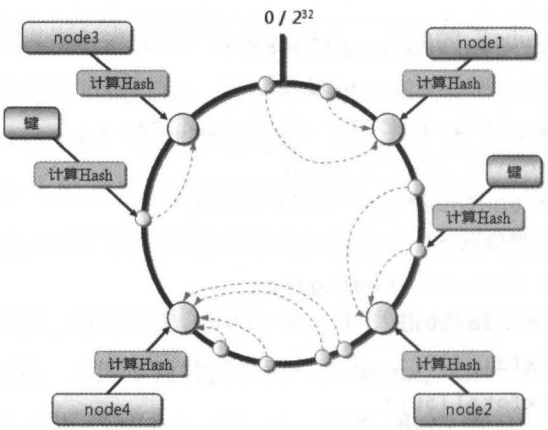


图 4-5 一致性 Hash 算法

随着 Redis 正式开始提供 Cluster 功能，对于运维人员和开发人员来说，集群节点的扩容将变得更加容易。简单来说，Redis 集群节点中一共包含 16384 个 Slot，不同的 Redis 节点内各自负责维护一小段 Slot 用于存储不同区间的数据。当客户端进行数据的读/写操作时，通过观察 Jedis 的源码我们可以发现，Key 作为路由条件会先进行 CRC16 运算，然后再 mod 16384，以此路由到指定 Redis 节点的 Slot 区间内，如下所示：

```
public static int getSlot(String key) {
key = JedisClusterHashTagUtil.getHashTag(key);
// optimization with modulo operator with power of 2
// equivalent to getCRC16(key) % 16384
return getCRC16(key) & (16384 - 1);
 }
```

在 Redis 非集群的情况下，开发人员在程序中可以直接通过 JedisPool 的 getResource()方法获取到一个 Jedis 实例。但是当 Redis 集群后，程序中则需要使用 JedisCluster 来实现数据的 CRUD 操作，如下所示：

```
private static JedisPoolConfig cfg;
private static JedisCluster jedisCluster;
private static String HOST = "127.0.0.1";
private static int PORT = 6379;
public static @BeforeClass void init() {
    /* 省略配置连接池信息代码 */
    jedisCluster = new JedisCluster(new HostAndPort(HOST, PORT), cfg);
}
public @Test void testJedis() {
    jedisCluster.set("key1", "value1");
    jedisCluster.get("key1");
}
```

在上述程序中，客户端代码通过 HostAndPort 来指定集群中的单个 Redis 节点信息。尽管我们可以声明一个 List<HostAndPort>来存储所有的 Redis 节点信息，但是只配置一个可用且有效的 Redis 节点信息也是可行的；因为当 Jedis 客户端连接上集群中的某一个 Redis 节点时，就意味着已经成功连接上了整个 Redis 集群环境。

4.3 同一热卖商品高并发读需求

即将经历“双 11”、“双 12”这种类型和规模的全民网购活动的开发人员必然是夜不能寐、如坐针毡的。假设你负责的应用系统能够有效应对这种级别的用户流量，甚至在最后一波访问高峰来临时都没有倒下，那么这必然会为你的整个职业生涯带来无以言喻的成就感和自豪感，并且这也会成为你引以为傲的宝贵经验。

笔者以限时抢购这个业务场景为例。在程序中使用缓存技术，不仅能够提升系统整体的响应速度，还能在一定程度上有效降低关系型数据库的负载压力。尽管对 Redis 进行 Cluster 后可以理论上认为容量是无限的，并且数据的读/写操作经过了水平化处理，不同的 Key 均会落到不同的缓存节点上，以此避免所有的用户流量都集中落到同一个缓存节点上。但是对于限时抢购场景下的热卖商品来说，由于单价比平时更给力、更具吸引力，那么自然会比平时吸引更大的流量进来，这时同一个 Key 必然会落到同一个缓存节点上，因此分布式缓存在这种情况下一定会出现单点瓶颈。笔者举一个贴近生活的例子，2016 年 8 月 14 日凌晨，中国著名演员王某在其个人微博上发布的关于其娇妻马某婚内出轨的消息，深深震撼了神州大地上的广大“吃瓜群众”，并且事件迅速发酵，并在接下来的一周内稳坐各大新闻版面头条。单从技术细节上来分析，假设这条消息是存储在分布式缓存中，但由于消息是缓存在固定的 Slot 上的，尽管缓存系统单点支撑 10 万/秒的 QPS 没有任何压力，但是在短短几分钟内，光是用户留言就达到了上百万条，更别说是阅读这条消息时所产生的流量，

因此分布式缓存的单点容量容易瞬间被撑爆，导致资源连接耗尽。为了解决分布式缓存可能存在的单点瓶颈，本书提供如下两种解决方案：

- 基于 Redis 集群多写多读方案；
- LocalCache 结合 Redis 集群的多级 Cache 方案。

4.3.1 Redis 集群多写多读方案

在默认情况下，假设某一个热卖商品的 Key 为“itemid123”，经过路由后，数据会被存储到集群环境中的某一个缓存节点上。我们把多写和多读这两个步骤拆分开来看，多写就是指在限时抢购活动正式开始之前，先对某一个热卖商品的 Key 进行加工和计算。假设集群环境中有 n 个缓存节点，那么加工和计算后的 Key 也应有 n 个，在最理想的情况下，每一个 Key 都应该被路由到一个指定的缓存节点上，其实多写操作就是为了实现数据的冗余存储，如图 4-6 所示。

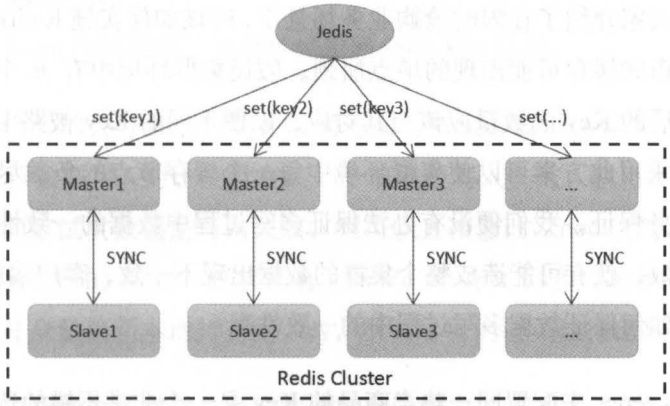


图 4-6 多写数据冗余存储

数据冗余存储实施起来是比较简单的，就是将之前的单写改为多写。由于 Redis 的处理能力比较强悍，同步多写并不会对程序的吞吐量造成太大的影响，当然如果

在程序中采用异步写的方式会更好。

一旦同一份商品数据被成功冗余存储到所有的缓存节点上，无论用户流量有多大，在并发环境下，客户端都可以根据加工后的 Key，采用轮询或随机等方式实现数据访问。这时集群环境中每一个缓存节点的负载压力均是一样的（单个缓存节点的负载量=用户流量 / 缓存节点个数），从而可以有效避免峰值流量时，某一个缓存节点的 OPS 特别高。

在此需要注意，多写多读方案并非完美的，一旦开始实施此方案，开发人员不得不思考两个问题。首先，如何保证多写过程中数据的一致性；其次，由于实现多写的 Key 是提前准备好的，一旦后期对集群环境进行了动态扩容，由于每一个缓存节点持有的 Slot 区间发生了变化，那么所有的 Key 都需要重新进行加工和计算。

4.3.2 保障多写时的数据一致性

笔者曾为大家介绍了在限时抢购业务场景下，应该如何实施 Redis 集群多写多读方案来解决分布式缓存可能出现的单点瓶颈。假设集群环境中有 n 个缓存节点，那么加工和计算后的 Key 的数量应该与其对应，以便不同的 Key 被路由到不同的缓存节点上。尽管采用此方案可以使集群环境中每一个缓存节点的负载尽可能均衡，但是由于缺少事务保证，我们便没有办法保证多写过程中数据的一致性；因此一旦某个节点写入失败，就有可能造成整个集群的数据出现不一致，客户端就会出现脏读。那么我们应该如何保证数据多写过程中的一致性呢？

使用 ZooKeeper 来配置同一热卖商品的 Key 是一个非常不错的选择，此时客户端对其进行监听，一旦 Watch 到 Znode 发生变化，所有的客户端全量更新本地持有的 Key 即可，如图 4-7 所示。

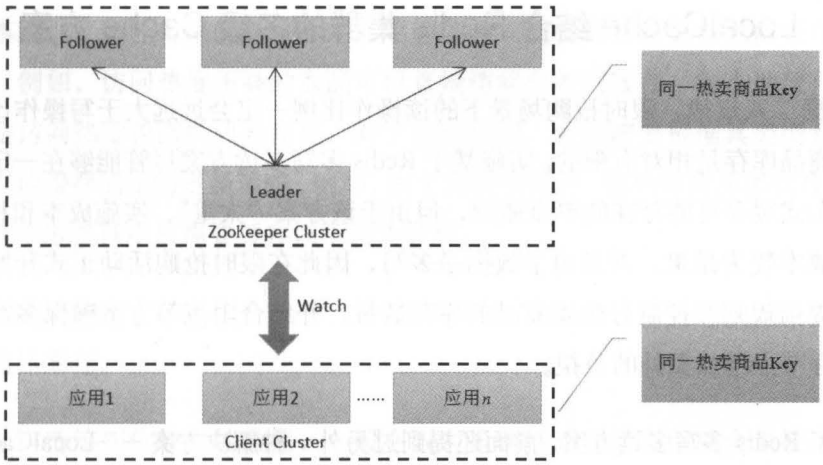


图 4-7 将 Key 配置在 ZooKeeper 中

下述三种情况会造成数据多写过程中产生失败：

- 网络环境发生抖动；
- Master/Slave 切换；
- Master/Slave 全部宕机，集群环境重新分配 Slot 区间。

上述三种情况都会导致客户端在数据写入时产生失败。当客户端使用加工和计算后的 Key 进行数据写入时，一旦在某一个节点处写入失败，就可以 Failover 几次；如果超过规定的重试次数还是写入失败，则可以直接修改 Znode 节点中配置的 Key 信息（剔除写入失败的 Key），这样所有监听 ZooKeeper 的客户端都会感知到 Znode 发生了变化，并全量更新本地持有的 Key，从而尽可能避免数据出现脏读。

当某一个节点发生故障并且客户端成功剔除指定的 Key 后，同一热卖商品剩下的 Key 的数量则为 $n-1$，也就是说，客户端接下来能够 Set/Get 的集群节点数量为 $n-1$。当然，在后续收到监控系统的告警后，可以修改 ZooKeeper 中配置的 Key，这样一来可操作的缓存节点数量即可恢复正常。

4.3.3 LocalCache 结合 Redis 集群的多级 Cache 方案

根据二八定律，限时抢购场景下的读操作比例一定会远远大于写操作比例，毕竟热卖商品库存是相对有限的。实施基于 Redis 多写多读方案尽管能够在一定程度上解决分布式缓存可能存在的单点瓶颈，但由于该方案“太重”，实施成本和相关 Key 的维护成本较为昂贵，并且由于数据是多写，因此在限时抢购活动正式开始之前，一定要提前规划和控制好热卖商品的库存数量，并结合限流等方案确保多写操作不会给缓存节点增加额外的负担。

除了 Redis 多写多读方案，前面还提到过另外一种解决方案——LocalCache 结合 Redis 集群的多级 Cache 方案。笔者在实际的生产环境中，也是使用此方案来解决限时抢购场景下分布式缓存可能存在的单点瓶颈。放眼整个互联网领域，LocalCache 与分布式缓存的结合必然能够为系统的后端存储带来更大的保障，如图 4-8 所示。

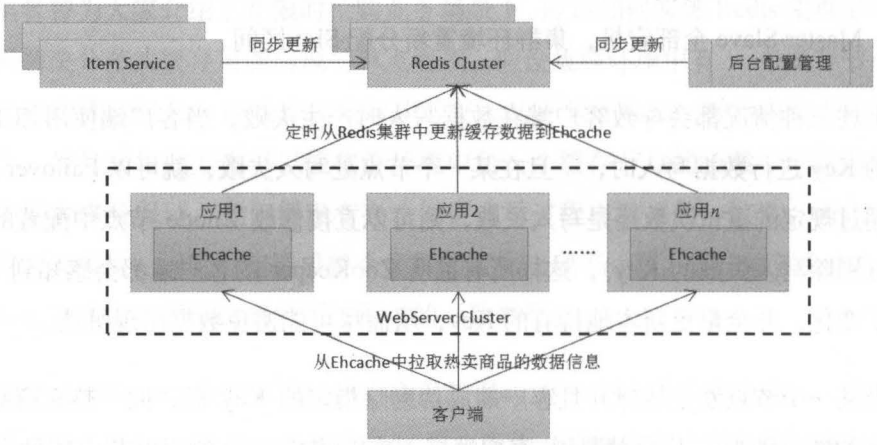


图 4-8 多级 Cache 示例

以 Ehcache 为例，由于本地缓存会共享同一个 JVM 进程内的 heap 空间，因此为

了避免本地缓存可能带来的种种弊端，笔者不建议把所有的商品信息都缓存在本地缓存中。例如，访问热度不高的商品可以直接访问分布式缓存，而本地缓存中存储更多的是访问热度较高的热卖商品。对于商品数据，我们需要根据其类型有针对性地配置本地缓存的定时更新策略。例如，由于商品的图片、视频等资源都缓存在 CDN 中，因此本地缓存中只需要缓存以下两类数据：

- 商品详情；
- 商品库存。

商品详情等变化较少的数据，一般在限时抢购活动开始之前就全量推送到所有参与限时抢购 Web 服务器的本地缓存上，直至活动结束。我们也不可能完全保证商品详情数据自始至终都不会发生变化，所以针对这类数据可以设置较长的定时轮询时间。

由于商品详情数据的变化频率较低，因此定时轮询时间可以设置得较长；但是商品库存等数据变化非常频繁，就需要将其定时轮询时间设置得较短，一般几秒后就可以从分布式缓存中拉取最新的库存数据。当大家看到这里时是否会产生一个疑问，本地缓存中存储的商品库存与实际商品库存之间可能会因为时差而造成数据的不一致，这样是否会导致超卖？对于读场景而言，其实完全可以接受在一定程度上出现数据脏读，因为这只会导致一些原本已经没有库存的少量下单请求误以为还有库存而已，等到最终扣减库存时再提示用户所购买的商品已经售罄即可。

在此需要注意，如图 4-8 所示，在实施多级 Cache 方案时，可以设置定时轮询时间，定时从分布式缓存中拉取最新的缓存数据更新到本地缓存中。但是千万不要为本地缓存设置 TTL 策略，因为这种做法将会产生一个问题：当用户流量过大时，一旦本地缓存的 TTL 过期，由于本地缓存中没有数据，Web 服务器便会直接从分布式缓存中拉取数据并更新到本地缓存上。试想一下，如果大量请求都无法在本地缓存命中，那么这些请求便会全部落到分布式缓存上，然后大量的更新请求会对本地

缓存重复写入，这会导致程序的吞吐量严重下降，尤其是变化频率较大的商品库存数据。

如果想缩短本地缓存与分布式缓存之间数据不一致的窗口期，本书还提供了一种准实时的多级 Cache 方案——引入消息队列，如图 4-9 所示。当我们通过后台配置管理或者商品服务修改了分布式缓存中存储的商品数据后，再把消息写入到消息队列中，所有订阅了目标 Topic 的消费者都可以消费到由生产者推送的商品数据。但实施这种方案会相对比较复杂，而且由于引入了消息队列，增加了额外的工作成本，以及分布式环境下外围系统的宕机风险。

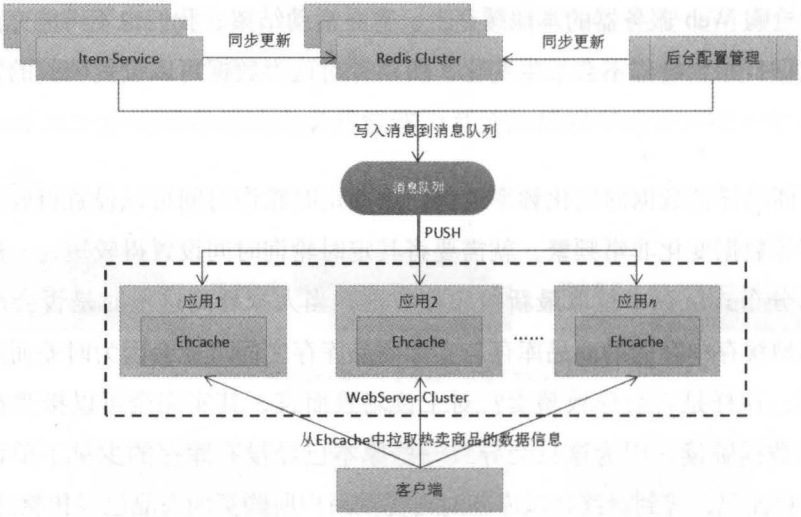


图 4-9 准实时的多级 Cache 的示例

4.3.4 实时热点自动发现方案

由于 SKU 非常多，因此任何一个电商平台都不会把所有的 SKU 数据全量存储在本地缓存中，只会对热点数据做优化。无论是采用 Redis 多写多读方案，还是选择多级 Cache 方案，都会面临一个问题，那就是如何配置热点 Key（HotKey）。笔者在

实际的生产环境中选择了将 HotKey 配置在集中式资源配置中心内，主要是为了方便管理热点数据；只不过在限时抢购活动正式开始前，开发人员需要从运营同学那里获取到所有热卖商品的 HotKey，否则一切都是徒劳无功的。

活动正式开始后，运维人员通过监控系统可以发现分布式缓存的单点容量处于比较平稳的水位，不会因为流量的影响而导致负载过大。但是希望大家记住，任何事情都不会一帆风顺，暴风雨的来临总是悄无声息的。热卖商品的 HotKey 尽管可以在活动开始前就提前分析出来，并由运营人员交给开发人员进行配置；但是那些提前发现不了并突然成为热点的数据，以及被热点数据瞬间附带起来的流量似乎就成了漏网之鱼。对于这种在运行时突然形成的热点，我们需要引入一种实时热点自动发现机制来进行热点保护。

目前，一些大型的互联网电商企业内部都搭建了适用于自身业务特点的实时热点自动发现平台。简单来说，我们可以将交易系统产生的相关数据，以及在上游系统中埋点上报的相关数据异步写入到日志系统中，然后通过实时热点自动发现平台对收集到的日志数据做调用次数统计和热点分析；数据符合热点条件后，就立即通知交易系统做好热点保护，防患于未然，如图 4-10 所示。

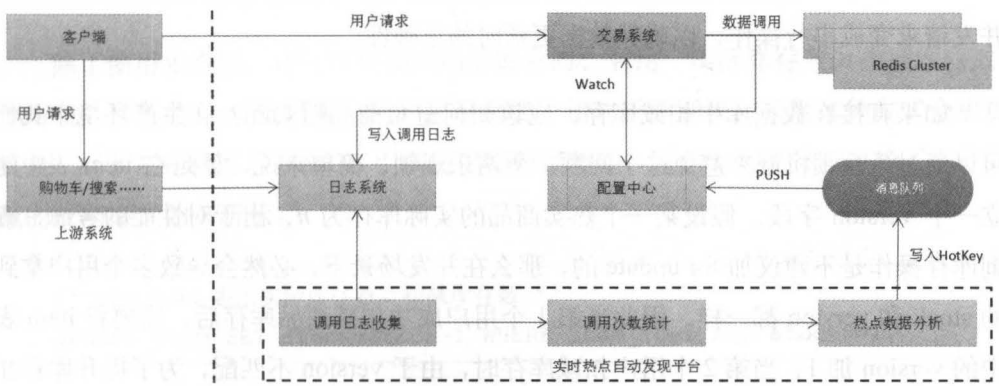


图 4-10 实时热点自动发现平台

4.4 同一热卖商品高并发写需求

在限时抢购场景下，为了避免分布式缓存可能存在的单点瓶颈，笔者建议大家在实际的生产环境中实施多级 Cache 方案。除了读需求，并发扣减同一热卖商品库存的写需求也是一件棘手的事情。

开发人员往往会将商品的真实库存存储在关系型数据库中，但大量的并发更新热点数据都是针对同一行的（本书以 MySQL 为例），那么这必然会引起大量的线程相互竞争 InnoDB 的行锁；并发越大等待的线程就越多，这会严重影响数据库的 TPS，导致 RT 线性上升，最终可能引发系统出现雪崩。为了避免数据库沦为瓶颈，我们可以将热卖商品库存的扣减操作转移至关系型数据库外或者合理控制并发写的流量。

4.4.1 InnoDB 行锁引起数据库 TPS 下降

在大部分情况下，商品库存都是直接在关系型数据库中进行扣减，那么在限时抢购活动正式开始后，那些单价比平时更给力、更具吸引力的热卖商品大家肯定都会积极踊跃地参与抢购，这必然会产生大量针对数据库同一行记录的并发更新操作。因此数据库为了保证原子性，InnoDB 引擎默认会对同一行数据记录加锁，把前端的并发请求变成串行操作，以确保数据更新时的正确性。

如果直接在数据库中扣减库存，应该如何避免商品超卖呢？在生产环境中我们可以通过乐观锁机制来避免这个问题。所谓乐观锁，简单来说，就是在 item 表中建立一个 version 字段。假设某一个热卖商品的实际库存为 n，出于对性能的考虑，查询库存操作是不建议加 for update 的，那么在并发场景下，必然会导致多个用户拿到的 stock 和 version 都一样。因此当第 1 个用户成功扣减商品库存后，需要将 item 表中的 version 加 1；当第 2 个用户扣减库存时，由于 version 不匹配，为了提升库存扣减的成功率，可以适当进行重试，如果库存不足，则说明商品已经售罄，反之扣减

库存后 version 继续加 1。关于在数据库中使用乐观锁扣减库存的伪代码，如下所示：

```
public void testStock() {
    if (version不一致时的重试次数阈值) {
   SELECT stock,version FROM item WHERE item_id=1;
        if (如果查询的指定商品存在) {
            if (判断stock是否够扣减) {
           UPDATE item SET version=version+1,stock=stock-1 WHERE
                       item_id=1 AND version="+ version +";
                if (扣减库存失败) {
                    /* version不一致时开始尝试重试 */
                    testStock();
                } else {
                    logger.info("扣减库存成功");
                }
            } else {
                logger.warn("指定商品已售罄");
            }
        }
    }
}
```

除了使用乐观锁，还可以在扣减商品库存时，利用“实际库存数≥扣减库存数”作为条件来替代 version 匹配，防止商品超卖。相对于乐观锁，采用这种方式会更加直接，由于充分利用了 InnoDB 引擎提供的行锁特性，因此大大提升和保障了库存扣减的成功率，如下所示：

```
/* stock≥1表示实际库存数≥扣减库存数 */
UPDATE item SET stock=stock-1 WHERE item_id=1 AND stock>=1;
```

本书示例的上述两种方案均能够有效避免商品超卖，但在并发较大时，由于大

量线程相互竞争 InnoDB 行锁时所引起的一系列问题，我们则不能忽视。在 MySQL 客户端中，可以使用如下命令查询出队列中等待拿锁的线程：

```
SELECT * FROM information_schema.INNODB_TRX WHERE trx_state='LOCK
WAIT';
```

4.4.2 在 Redis 中扣减热卖商品库存方案

如果系统前端不配合做限流消峰等处理，放任大量的并发更新请求直接在数据库中扣减同一热卖商品的库存数据，这将导致线程之间相互竞争 InnoDB 的行锁。由于数据库中针对同一行数据的更新操作是串行执行的，那么某一个线程在未释放锁之前，其余的线程将会全部阻塞在队列中等待拿锁，并发越高等待的线程就越多，这会严重影响数据库的 TPS，从而导致 RT 线性上升，最终可能引发系统出现雪崩。

InnoDB 的行锁特性其实是一把利与弊都很明显的双刃剑，在保证原子性的同时降低了可用性。那么应该如何保证大并发更新热点数据不会导致数据库沦为瓶颈，这其实是秒杀、抢购场景下最核心的技术难题之一。笔者在前面曾经提及过，可以尝试将热卖商品的库存扣减操作转移至数据库外。由于 Redis 的读/写能力远胜过任何类型的关系型数据库，因此在 Redis 中实现库存扣减是一个不错的替代方案。这样数据库中存储的商品库存可以被理解为实际库存，而 Redis 中存储的商品库存则为实时库存。

但在并发较大的情况下，直接在 Redis 中扣减库存一定会导致商品出现超卖现象，那这种情况应该如何避免呢？这就是本节的重点，引入分布式锁来避免超卖。当然分布式锁自身必须满足以下三点要求：

- 在任何情况下分布式锁都不能沦为系统瓶颈；
- 不能产生死锁；

- 支持锁重入。

分布式锁的实现方式较多，目前市面上比较常见的有两种，分别是基于 ZooKeeper 和 Redis 实现的分布式锁。出于对性能方面的考虑，本书仅以 Redis 实现的分布式锁为例。究竟应该如何在 Redis 中实现分布式锁呢？使用 Redisson 是一个不错的方案，或许习惯了使用 Jedis 的开发人员对 Redisson 有些陌生。相比于 Jedis，Redisson 确实算得上一款崭新的 Redis 客户端 API，它支持丰富的数据类型，并且是线程安全的，底层还使用了 Netty 4 进行网络通信。那么我们能否在程序中用 Redisson 代替 Jedis 来与 Redis 进行交互呢？其实，Redisson 诞生之初，从本质上来说仅仅是为了扩展 Jedis 的部分功能，两者是并存的，比如 Redisson 并不支持 String 类型的数据结构。

本书所使用的 Redisson 版本为 2.2.11，如下所示：

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>2.2.11</version>
</dependency>
```

成功下载好运行 Redisson 所需的相关构件后，笔者开始演示如何在程序中使用 Redisson 客户端实现基于 Redis 的分布式锁，如下所示：

```
private static Config config;
private static ClusterServersConfig clusterServersConfig;
private static String address = "127.0.0.1:6379";
public static @BeforeClass void init() {
    config = new Config();
    /* 使用集群模式 */
    clusterServersConfig = config.useClusterServers();
    clusterServersConfig.addNodeAddress(address);
```

```
        clusterServersConfig.setMasterConnectionPoolSize(100);
        clusterServersConfig.setSlaveConnectionPoolSize(100);
        clusterServersConfig.setTimeout(1000);
    }
    public @Test void testLock() {
        RedissonClient redisson = null;
        try {
            redisson = Redisson.create(config);
            RLock lock = redisson.getLock("testLock");
            /* 获取锁 */
            lock.lock(20, TimeUnit.MILLISECONDS);
            /* 释放锁 */
            lock.unlock();
            /* 尝试获取锁 */
            boolean result = lock.tryLock(10, 20, TimeUnit.MILLISECONDS);
            /* 判断是否成功获取到锁 */
            if (result) {
                lock.forceUnlock();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (null != redisson) {
                redisson.shutdown();
            }
        }
    }
```

在上述程序中，演示了 Redisson 提供的两种获取分布式锁的方法。lock（long leaseTime, TimeUnit unit）方法中的第 1 个参数用于设定分布式锁的租约时间，而第 2 个参数则为时间单位。使用这种方式意味着在某一个获取到锁的线程未释放锁之

前，其他线程只能够在队列中阻塞等待，这和 InnoDB 引擎提供的行锁机制如出一辙，并发越高等待的线程越多。因此，在并发较大的情况下，建议大家使用 tryLock（long waitTime, long leaseTime, TimeUnit unit）方法获取分布式锁。

在 tryLock()方法中开发人员可以通过参数“waitTime”来设定获取分布式锁的等待时间，超出规定的时间阈值后，线程将不再继续等待拿锁；那么为了提升库存扣减的成功率，可以在获取锁失败后尝试多次。相比 lock()方法的拿锁方式，后者在并发较大的情况下不会使分布式锁沦为系统瓶颈，但是商品库存的扣减成功率会受到一定影响。

将商品库存的扣减操作转移至 Redis 中主要是为了避免数据库沦为系统瓶颈。既然性能问题得到了解决，那么变化后的实时库存应该如何同步到数据库中呢？当系统获取到分布式锁并成功扣减 Redis 中的实时库存后，可以将消息写入到消息队列中，由消费者负责实际库存的扣减。由于采用了排队机制，并发写入数据库时的流量可控，因此数据库的负载压力就会始终保持在一个恒定的范围内，不会因为流量的影响而导致数据库性能下降，如图 4-11 所示。

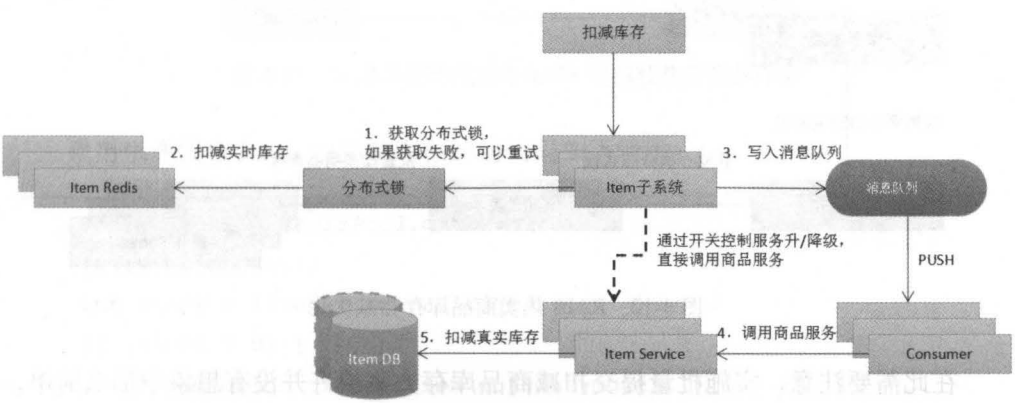


图 4-11 在 Redis 中实施热卖商品的库存扣减

4.4.3 热卖商品库存扣减优化方案

为了避免商品超卖，无论是直接在数据库中扣减库存，还是在 Redis 中扣减库存，都必须依赖于串行化和锁机制。假设商品库存数量为 n，那么锁的获取次数也为 n。大家思考一下，如何有效减少锁的获取次数来达到提升系统整体 TPS 的目的？或许批量提交扣减商品库存是一个不错的优化方案。阿里开源的 MySQL 分支 AliSQL 数据库已经支持了这项特性，大家可以直接阅读 4.4.5 节的内容。那么在 Redis 中应该如何优化热卖商品的库存扣减操作呢？

简而言之，当前端发起库存扣减请求后，可以先对这些请求展开收集；假设收集次数被设定为 100 次，那么达到阈值后再对这些请求做合并处理，获取一次分布式锁后就一次性提交到 Redis 中进行库存扣减，如图 4-12 所示。这会存在一种特殊的情况，即商品库存不足扣减时，这一批库存扣减操作都将失败。采用这种方式最大的好处就是可以将原先的串行化操作变成批处理操作，大大提升了系统整体的 TPS。假设某一个热卖商品的库存数量为 1000，收集次数被设定为 100，那么总共只需要获取 10 次（商品库存数量/收集阈值）分布式锁即可。

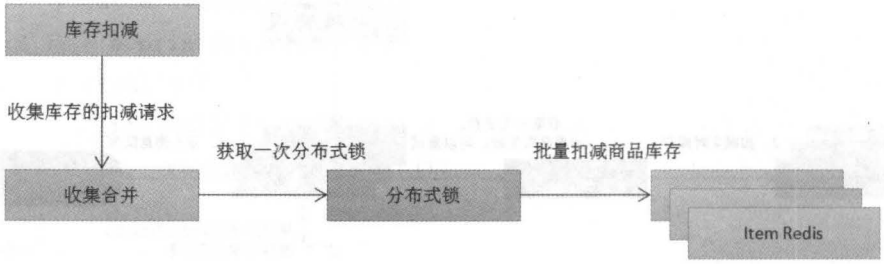


图 4-12 Redis 热卖商品库存扣减优化

在此需要注意，实施批量提交扣减商品库存方案或许并没有想象中那么简单，此方案与业务结合得过于紧密，因此在实施过程中有以下两个问题需要慎重考虑和尽早规划：

- 库存扣减结果应该如何响应给指定的用户；
- 如何避免商品库存售不罄。

在 Redis 中扣减热卖商品的库存，除了可以使用分布式锁来保证数据的一致性，我们还有其他更优的解决方案，毕竟分布式锁存在较大的性能开销，因此为了能够更好地提升系统的吞吐量，笔者建议大家在生产环境中可以考虑采用 Redis 提供的 Watch 命令来实现乐观锁。和 4.4.1 节中笔者为大家介绍的基于 MySQL 的乐观锁机制一样，并发环境下，通过 Watch 命令对目标 Key 进行标记后，当事务提交时，如果监控到目标 Key 对应的值已经发生了改变，那么就意味着版本号发生了改变，因此这一次的事务提交操作就需要进行回滚，如图 4-13 所示。

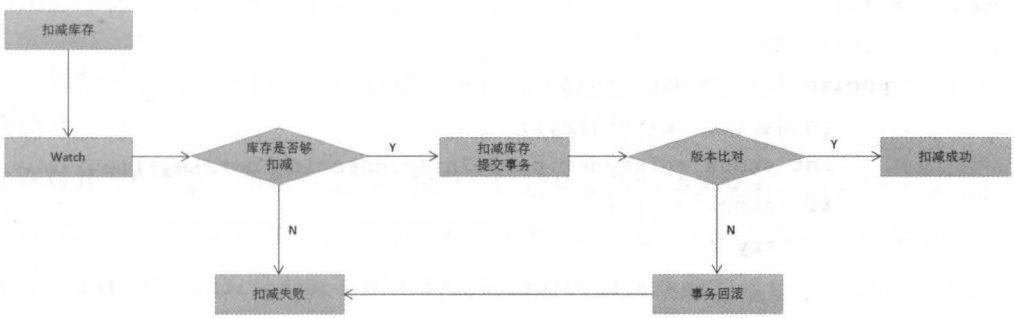


图 4-13 采用乐观锁机制在 Redis 中扣减热卖商品库存

采用乐观锁在 Redis 中扣减库存的伪代码如下所示：

```
Jedis jedis = jedisPool.getResource();
jedis.watch(key);
int stock = Integer.parseInt(jedis.get(key));
if (stock > 0) {
    Transaction transaction = jedis.multi();
    Transaction.decr(key);
    List<Object> result = transaction.exec();
    if (result != null && !result.isEmpty()) {
```

```
            System.out.println("抢购成功");
        }
        jedis.unwatch();
    }
    jedisPool.returnResource(jedis);
```

在此大家需要注意，如果使用的是 Redis3.x 以上版本，而且 Redis 采用 Cluster 模式进行部署，但 JedisCluster 目前并不支持 Watch 及事务方法，当然这并不意味着集群环境下我们无法使用乐观锁，我们可以重写其 Set 方法，利用它默认的路由机制来实现 Watch 命令，如下所示：

```
    String result = new JedisClusterCommand<String>(connectionHandler,
maxAttempts) {
        @Override
        public String execute(Jedis connection) {
            connection.watch(key);
            int stock = Integer.parseInt(connection.get(key));
            if (stock > 0) {
                try {
                    Transaction transaction = connection.multi();
                    Transaction.decr(key);
                    List<Object> result = transaction.exec();
                    return result != null && !result.isEmpty() ? "true" :
"false";
                } finally {
                    connection.unwatch();
                }
            }
            return "false";
        }
    }.run(key);
```

4.4.4 控制单机并发写流量方案

除了可以将库存的扣减动作从关系型数据库中转移至 Redis 中进行扣减，本书还提供了另外两种改造成本相对较低的解决方案，如下所示：

- 单机排队串行写方案；
- 抢购限流方案。

在应用层实现排队机制，将热卖商品的库存扣减操作设置队列串行执行，这样便可以降低大促场景下针对同一热卖商品并发写的流量，虽然对于集群环境中的单个节点而言，库存的扣减操作是串行执行的，但是从整体上来看仍然是并发执行的，由集群环境中的节点数量决定并发写的流量。

但是单机排队串行写的执行效率似乎并不高效，尽管数据库中针对同一行数据的并发写操作多线程之间需要相互竞争 InnoDB 的行锁，只有获取到锁的线程才允许对其进行写入。虽然也是串行执行，但是从压测的结果来看，前者的效率远低于后者，因此在生产环境中，不建议采用这种解决方案。笔者曾为大家讲解了如何通过计数器算法来实现大促场景下的抢购限流，如果系统前端能够配合交易系统做好限流保护，控制并发写的流量，那么必然能够有效降低数据库的负载压力，提升 TPS，同时还能够有效避免同一热卖商品占用较多的数据库连接资源，如图 4-14 所示。

抢购限流其实也是一种排队机制，和串行写方案不同的是，单机也允许并发，因此可以大大提升库存扣减时的执行效率，并且由于并发写的流量可控，可以让数据库的负载处于一个比较均衡的水位，不会因为峰值流量过大，导致系统被无情击垮。如果抢购限流方案能够结合 AliSQL 数据库来共同应对大促场景，那么系统的整体性能将得到不小的提升。

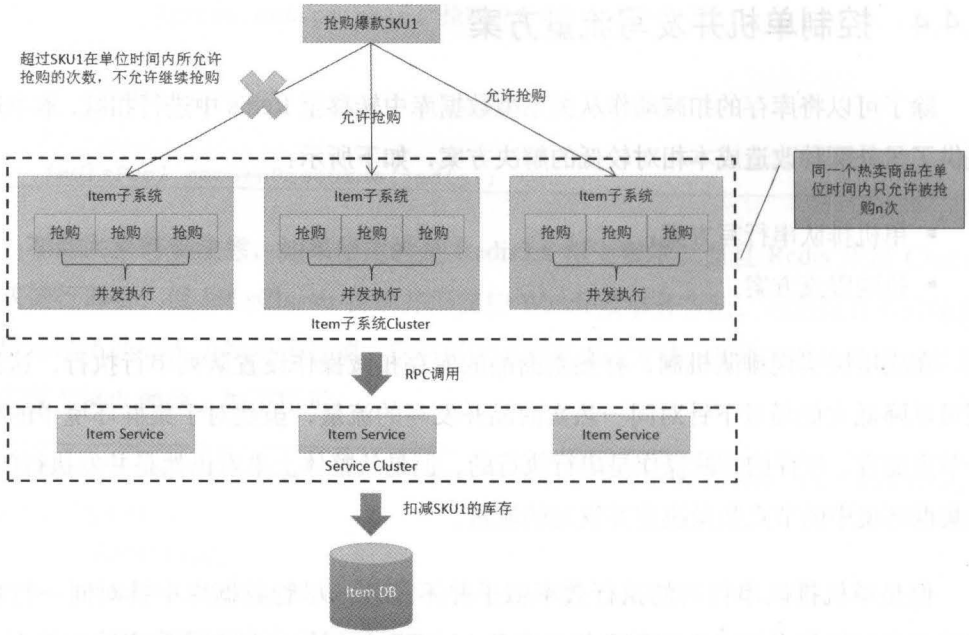


图 4-14 抢购限流控制并发写的流量

4.4.5 使用阿里开源的 AliSQL 数据库提升秒杀场景性能

2016 年 8 月 9 日，阿里正式启动了 AliSQL 数据库的开源计划。从官方发布的测试结果来看，AliSQL 数据库的综合性能较 MySQL 官方版本提升了约 70%，尤其是针对秒杀场景做了特殊优化，性能可提升 100 倍左右。如果仅仅是希望使用 AliSQL 替换 MySQL 官方版本来应对秒杀场景，那么可以将 AliSQL 理解为一款为互联网电商业务特殊定制的高性能 MySQL 数据库，AliSQL 在云计算及金融等领域也能够大展拳脚。

此次开源的 AliSQL 所带来的优化特性较多，本书仅以秒杀场景为例。前面曾经提及过，采用批量提交扣减商品库存方案能够有效提升系统整体的 TPS，那么这在 AliSQL 中又是如何实现的呢？其实 AliSQL 在 InnoDB 引擎层前面对同一行的热点更

新 SQL 做了收集。简而言之，AliSQL 首先会对热点记录做 Hash，其中每一个桶就是一个热点行，对每个桶做收集，然后由桶里面的第一个事务一批次地把当前桶里收集的所有库存扣减请求一次性提交掉，这样就将原先的串行化操作变成了批处理操作。为了提升整体性能，当第一批提交的时候第二批开始收集，这样就将单线程的串行变成了批处理的线程，如图 4-15 所示。无论是在 Redis 中扣减库存，还是在 AliSQL 中扣减库存，为了提升系统整体的 TPS，都是先收集库存扣减请求，达到阈值后做合并处理，最后批量进行提交。

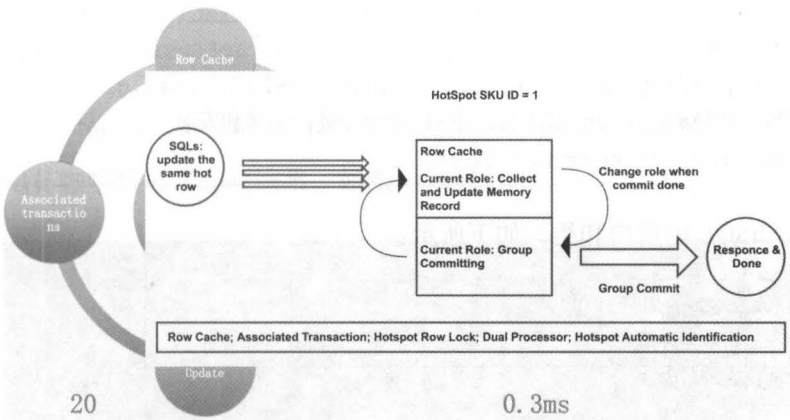


图 4-15 AliSQL 热卖商品库存扣减优化

AliSQL 于 2016 年 9 月开始提供内测，真正的开源日期是 10 月，笔者当时有幸成为 AliSQL 的第一批实验“小白鼠”。目前 GitHub 上已经有两个 RELEASE 版本，AliSQL 的项目地址为 https://github.com/alibaba/AliSQL。接下来就为大家演示 AliSQL 的编译和安装过程，在此之前，要先准备一些基础环境，笔者的操作系统版本为 CentOS7.0，所使用的 AliSQL 版本为 AliSQL 5.6.32，也可以下载其他版本的 AliSQL，但是为了避免在编译和安装过程中发生一些不必要的错误，建议使用和本书一致的 AliSQL 版本。

首先安装一些编译 AliSQL 源码所必需的编译器和库，如下所示：

```
Yum install gcc gcc-c++ ncurses-devel perl
```

编译和安装 cmake，本书使用的 cmake 版本为 2.8.10.2，如下所示：

```
#下载 cmake
wget https://www.cmake.org/files/v2.8/cmake-2.8.10.2.tar.gz
#解压后，切换到/cmake-2.8.10.2 目录下对 cmake 进行编译和安装
./bootstrap;make;make install
```

编译和安装 bison，本书使用的 bison 版本为 2.7.1，如下所示：

```
#下载 bison
wget http://ftp.gnu.org/gnu/bison/bison-2.7.1.tar.gz
#解压后，切换到/bison-2.7.1 目录下对 bison 进行编译和安装
./configure;make;make install
```

设置 AliSQL 的用户和组，如下所示：

```
#添加 AliSQL 的用户和组
groupadd mysql
#新增 AliSQL 用户
useradd -r -g mysql mysql
```

创建编译和安装 AliSQL 所需的目录，如下所示：

```
#创建 AliSQL 安装目录
mkdir -p /usr/local/mysql
#创建 AliSQL 数据库数据文件目录
mkdir -p /usr/local/data/mysqldb
```

一切准备就绪后，从 GitHub 上下载 AliSQL 的源码，如下所示：

```
https://github.com/alibaba/AliSQL/archive/AliSQL-5.6.32-2.zip
```

成功下载好 AliSQL 后，通过命令“unzip”对其进行解压，然后切换到目录

/AliSQL-AliSQL-5.6.32-2 下，设置 AliSQL 的一些编译参数，如下所示：

```
cmake
-DCMAKE_INSTALL_PREFIX=/usr/local/mysql
-DMYSQL_UNIX_ADDR=/usr/local/mysql/mysql.sock
-DDEFAULT_CHARSET=utf8
-DDEFAULT_COLLATION=utf8_general_ci
-DWITH_INNOBASE_STORAGE_ENGINE=1
-DWITH_ARCHIVE_STORAGE_ENGINE=1
-DWITH_BLACKHOLE_STORAGE_ENGINE=1
-DMYSQL_DATADIR=/usr/local/data/mysqldb
-DMYSQL_TCP_PORT=3306 -DENABLE_DOWNLOADS=1
```

使用命令“make”对 AliSQL 的源码进行编译，AliSQL 的编译过程比较耗时，大家可以喝杯咖啡后再来验收结果，如图 4-16 所示。

图 4-16 AliSQL 源码的编译过程

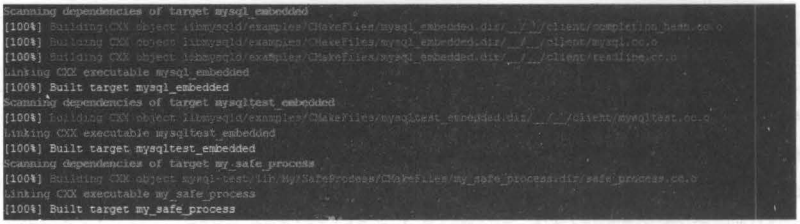


图 4-16 AliSQL 源码的编译过程（续）

如果编译之前的基础环境都已经准备完毕，那么 AliSQL 的编译过程应该是非常顺利的。完成编译后要做的事情就是使用命令“make install”对 AliSQL 进行安装，如图 4-17 所示。

```
[ 84%] Built target simple-t
[ 84%] Built target skip-t
[ 84%] Built target skip_all-t
[ 84%] Built target todo-t
[ 84%] Built target basic-t
[ 84%] Built target innochecksum
[ 84%] Built target my_print_defaults
[ 84%] Built target mysql_waitpid
[ 84%] Built target perror
[ 84%] Built target replace
[ 85%] Built target resolve_stack_dump
[ 85%] Built target resolveip
[ 85%] Built target mysql
[ 85%] Built target mysql_config_editor
[ 85%] Built target mysql_plugin
[ 85%] Built target comp_sql
[ 85%] Built target GenFixPrivs
[ 85%] Built target mysql_upgrade
[ 85%] Built target mysqladmin
[ 85%] Built target mysqlbinlog
[ 85%] Built target mysqlcheck
[ 85%] Built target mysqldump
[ 85%] Built target mysqlimport
[ 86%] Built target mysqlshow
[ 86%] Built target mysqlslap
[ 86%] Built target mysqltest
[ 86%] Built target bug25714
[ 86%] Built target mysql_client_test
[ 86%] Built target mysql_tzinfo_to_sql
[ 86%] Built target mysqld
[ 86%] Built target partition_embedded
[ 87%] Built target sqlgunitlib
[ 87%] Built target udf_example
[ 99%] Built target sql_embedded
[ 99%] Built target mysqlserver
[100%] Built target mysql_client_test_embedded
[100%] Built target mysql_embedded
[100%] Built target mysqltest_embedded
[100%] Built target my_safe_process
```

图 4-17 AliSQL 的安装过程

编译和安装工作完成后，修改 AliSQL 目录的所有者和组，如下所示：

```
#切换到/usr/local/mysql 目录下，修改 AliSQL 安装目录
chown -R mysql:mysql .
#切换到/usr/local/data/mysqldb 目录下，修改 AliSQL 数据库文件目录
chown -R mysql:mysql .
```

对已经安装完成的 AliSQL 进行一些初始化工作，如下所示：

```
scripts/mysql_install_db
--user=mysql  --datadir=/usr/local/data/mysqldb
```

复制 AliSQL 的服务启动配置文件到/etc 目录下的 my.cnf 文件中，如果目标文件已经存在则进行覆盖，如下所示：

```
cp /usr/local/mysql/support-files/my-default.cnf /etc/my.cnf
```

复制 AliSQL 的服务启动脚本到/etc 目录下，如下所示：

```
cp /usr/local/mysql/support-files/mysql.server /etc/init.d/mysqld
```

编辑文件/etc/profile 配置 AliSQL 的环境变量，如下所示：

```
PATH=/usr/local/mysql/bin:/usr/local/mysql/lib:$PATH
export PATH
```

当 AliSQL 的环境变量配置完成后，通过命令“source /etc/profile”让其立即生效，再通过命令“service mysqld start”启动 AliSQL 服务。AliSQL 启动后，通过命令“ps -aux|grep mysql”可以验证服务是否正常启动，如果启动成功，便可以使用 MySQL 客户端对其进行登录和访问操作。至此，AliSQL 的编译和安装过程就结束了，如果无法进行正常编译和安装，可以在 GitHub 上进行询问。

4.5 本章小结

热点数据的读/写操作其实是秒杀、抢购场景下最核心的技术难题。本章以为什么需要在系统中使用缓存技术作为开篇，重点讲解了本地缓存 Ehcache 及分布式缓存 Redis 技术，并介绍了如何使用 Jedis、Redisson 等客户端 API 与 Redis 进行交互。

当前期铺垫结束后，笔者切入了本章的主题，这也是笔者在实际工作中遇到的技术难题：限时抢购场景下的同一热卖商品高并发读/写需求。笔者首先在热点数据的读操作上，提供了基于 Redis 的多写多读方案，以及多级 Cache 方案。接下来在热点数据的写操作上，为大家介绍了如何将热卖商品的库存扣减操作转移至 Redis 中，以及其批量提交的优化方案。最后，笔者为大家介绍了阿里开源的 AliSQL 高性能关系型数据库，以及如何亲自动手编译和安装 AliSQL，结合限流手段和 AliSQL 来共同应对大促场景或许是一个不错的方案。

第5章 数据库分库分表案例

大型网站几乎时时刻刻都在接受着高并发和海量数据的洗礼，随着用户规模的线性上升，单库的性能瓶颈会逐渐暴露出来，由于数据库的检索效率越来越慢，导致生产环境中产生较多的慢速 SQL。对于非结构化的数据，可以将其存储在 NoSQL 数据库中来提升性能，但是重要的业务数据，仍然要落盘在关系型数据库（如 MySQL 数据库）中。那么如何提升关系型数据库的并行处理能力和检索效率就成为了架构师需要思考和解决的棘手问题，并且单库如果宕机，业务系统也就随之瘫痪了。因此，在互联网场景下，架构师务必要确保后端存储系统具备高可用性和高性能，为了解决这些问题，目前互联网场景下常见的做法便是对数据库实施分库分表，即 Sharding 改造。

笔者是在 2013 年正式踏入互联网这个领域的，也是从那个时候开始接触分库分

表的，不过当时的分库分表中间件并不多。笔者接触到的第一款分库分表中间件为淘宝的 TDDL（TAOBAO DISTRIBUTE DATA LAYER），但其 Matrix 层源码直至今日也并未开源实属遗憾。现在，除了 TDDL，我们有了更多的选择，Shark、MyCat 等优秀的分库分表中间件层出不穷。本章不仅会对数据库的架构演变过程进行重点讲解，还会实战演示 Shark 中间件的具体使用方式，以及数据库实施分库分表改造后所带来的影响的一系列解决方案。

5.1　关系型数据库的架构演变

在互联网场景下，关系型数据库常见的性能瓶颈主要有两个，如下所示：

- 大量的并发读/写操作，导致单库出现难以承受的负载压力；
- 单表存储数据量过大，导致检索效率低下。

5.1.1　数据库读写分离

任何一个刚上线的互联网项目的前期用户规模都不大，系统整体并发量相对较小，因此企业一般都会在这个阶段将所有的数据信息存储在单库中进行读/写操作。但是随着用户规模不断提升，单库逐渐力不从心，TPS/OPS 越来越低，因此到了这个阶段，DBA 会将数据库设置为读写分离状态（生产环境一般会采用一主一从或一主多从），由 Master 负责写操作，而 Slave 作为备库，不会开放写操作，但可以允许读操作，主从之间保持数据同步即可。根据二八法则，80%的数据库操作是读操作，剩下的 20%则为写操作。读写分离后，可以大大提升单库无法支撑的负载压力，如图 5-1 所示。在此需要注意，如果 Master 存在 TPS 较高的情况，Master 与 Slave 数据库之间数据同步是会存在一定延迟的，因此在写入 Master 之前最好将同一份数据落到缓存中，以避免高并发情况下，从 Slave 中获取不到指定数据的情况发生。

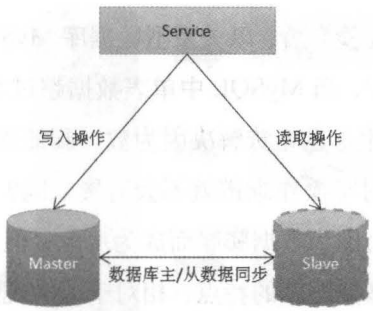


图 5-1 数据库读写分离

5.1.2 数据库垂直分库

前一节笔者为大家讲解了数据库的读写分离案例。由于将读/写操作进行了分离，Master 负责写入，Slave 负责读取，系统的整体吞吐量相对于单库来说自然有一定的提升，但是只依靠数据库的读写分离并不能一劳永逸，随着用户规模的线性上升，系统瓶颈一定会暴露。因此到了这个阶段，DBA 会开始对数据库进行垂直分库。所谓垂直分库，就是企业根据自身业务的垂直划分，将原本冗余在单库中的数据表拆分到不同的业务库中，实现分而治之的数据管理和读/写操作，如图 5-2 所示。

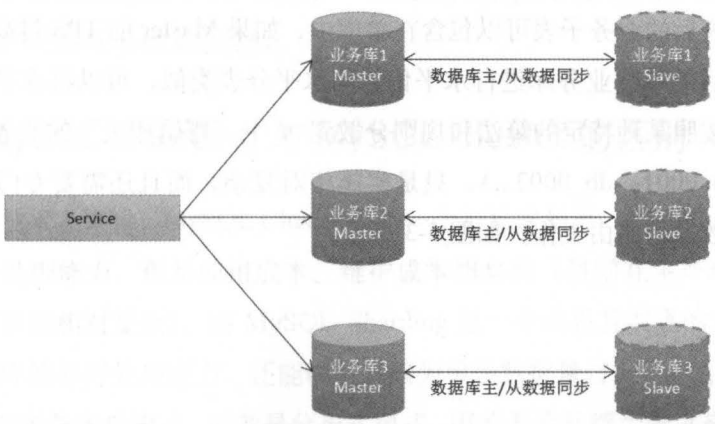


图 5-2 数据库垂直分库

结合笔者实际的生产经验来看，以关系型数据库 MySQL 为例，由于单一业务的数据信息仍然落盘在单表中，当 MySQL 中单表数据超过 500 万行时，读操作就会逐渐成为瓶颈，哪怕索引重建，也无法解决因为数据膨胀而带来的检索效率低下等问题。大家可能会有疑问，对写操作难道就不会有影响吗？由于写是顺序写，所以基本上数据库的写入操作不会因为数据膨胀而成为瓶颈，但是读操作一定会存在上限。这其实也是 RDBMS 等类型数据库的特点，相对于非常规的 NoSQL 数据库而言，由于双方的底层存储架构不同，所以自然无法等价。因此到了这个阶段，DBA 就需要在垂直分库的基础上实施水平分表。

5.1.3　数据库水平分库与水平分表

当大家对数据库的读写分离和垂直分库等概念有了一定的了解后，笔者再为大家介绍互联网场景下关系型数据库中应对高并发、单表数据量过大的最终解决方案。水平分表就是将原本冗余在单库中的单个业务表拆分为 *n* 个“逻辑相关”的业务子表（如 tab_0000、tab_0001、tab_0002...），不同的业务子表各自负责存储不同区间的数据，对外形成一个整体，这就是大家常说的 Sharding 操作。

水平分表后的业务子表可以包含在单库中，如果 Master 的 TPS 过高，则还可以对垂直分库后的单一业务库进行水平化。同水平分表类似，可以将水平分表后的这些业务子表按照某种特定的算法和规则分散到 *n* 个“逻辑相关”的业务子库中（如 db_0000、db_0001、db_0002...），只是实施相对复杂，而且还需要专门的 Sharding 中间件负责数据的路由工作，如图 5-3 所示。

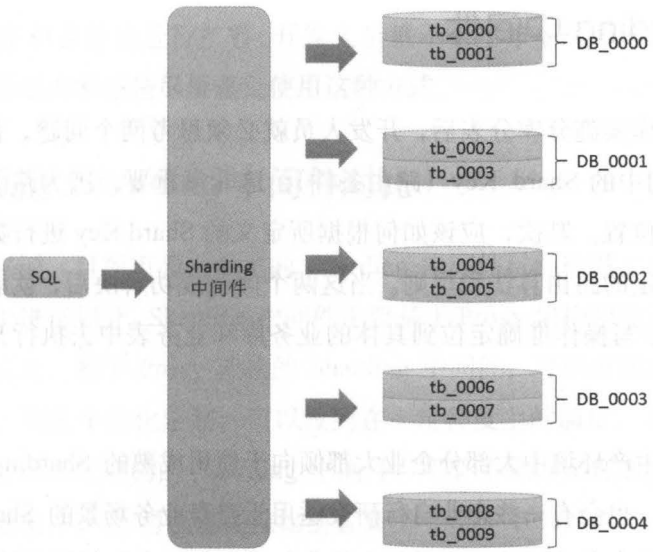


图 5-3 数据库分库分表

在生产环境中实施分库分表操作，主要是为了解决高并发场景下单库的性能瓶颈，并充分利用分布式的威力提升数据库的读/写性能。假设后续业务表中的数据量又一次达到存储阈值并对性能产生影响时，DBA 只需要再次对现有的业务库和业务表横向扩容，并迁移数据即可。目前，一些大型的互联网电商企业内部都对分库分表场景有着广泛的应用和实施经验。

5.1.4 MySQL Sharding 与 MySQL Cluster 的区别

单纯从技术上来讲，MySQL Cluster 只是一个数据库集群，其优势只是扩展了数据库的并行处理能力，但是使用成本、维护成本相当高（目前在生产环境中使用者较少，而且实施相对复杂）。而 MySQL Sharding 是一个成熟且实惠的方案，不仅可以提升数据库的并行处理能力，还能够解决因为单表数据量过大所产生的检索瓶颈。简而言之，前者是集群模式，后者是分布式模式，因此无论从哪个维度来看，Sharding 都是互联网当下最好的选择。

5.2 Sharding 中间件

一旦数据库实施分库分表后，开发人员就必须思考两个问题。首先，必须明确定义 SQL 语句中的 Shard Key（路由条件），这非常重要，因为路由维度直接决定了数据的落盘位置。其次，应该如何根据所定义的 Shard Key 进行数据路由，这需要定义一套特定的路由算法和规则。当这两个问题成功解决后，就能够让任何一条 SQL 语句的读/写操作准确定位到具体的业务库和业务表中去执行开发人员预期的操作。

目前，在生产环境中大部分企业大都倾向于使用成熟的 Sharding 中间件完成数据的路由工作，也会有一些企业自行研发适用于自身业务场景的 Sharding 中间件，不过当你所处的企业或团队，连最基本的业务进度都没办法保证且缺少实施经验时，笔者更推荐选择前者，毕竟成熟的中间件产品都是经过生产检验的，基本没有必要再花时间去踩坑和填坑。Sharding 中间件的领域模型定位如图 5-4 所示。

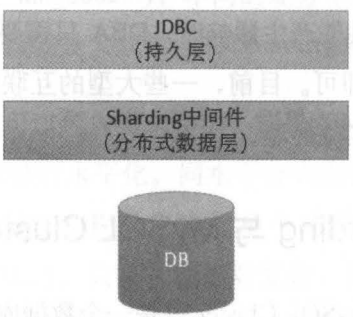


图 5-4 Sharding 中间件的领域模型定位

有一些 Sharding 中间件会将路由逻辑硬编码在持久层代码中，这样，不仅耦合度高，更重要的是不具备任何灵活性。确切地说，持久层根本不应该关心数据路由的工作，它的任务核心只是负责对数据库的 CRUD 操作，并且后续一旦 DBA 需要

对现有的业务库和业务表进行扩容，开发人员就不得不修改硬编码在持久层代码中的路由逻辑，所以大家还是尽量避免使用这种方式。

5.2.1 常见的 Sharding 中间件对比

相对早期来说，目前市面上常见的 Sharding 中间件已经称得上百花齐放了，并且从体系架构上进行划分，Sharding 中间件主要基于 Proxy 架构和应用集成架构两大类组成。简而言之，基于 Proxy 架构的 Sharding 中间件，可以灵活实现任意的关系型数据库协议，满足个性化定制，可以做到在一定程度上的通用，不局限于任何数据库。而基于应用集成架构的 Sharding 中间件，尽管不能够实现通用性需求，但是由于应用直连数据库，读/写性能往往比前者高出 10%～20%。总之，两种体系架构截然不同的 Sharding 中间件都各有优缺点，具体使用哪一种还需要结合自身实际的业务场景而定。

本书以基于应用集成架构的 Sharding 中间件为例。早期比较成熟的只有淘宝的 TDDL，但 TDDL 并非是完美的，比如社区活跃度低、技术文档资料匮乏，而且部分功能开源但核心功能闭源，因此注定了 TDDL 无法直接为欣赏它的非淘宝系用户服务。现在想要使用 TDDL 还有一种间接的方式，那就是淘宝已经将 TDDL 作为阿里云上的一个收费服务——分布式关系型数据库服务（Distribute Relational Database Service，DRDS）。开发人员无须关心底层的数据库存储架构，只需要使用阿里云提供的数据服务即可，这种方式有利有弊，最明显的弊端就是，后期如果企业自行构建数据中心时，数据迁移将是一件令人非常头痛的事情。

目前市面上常见的一些 Sharding 中间件产品对比如表 5-1 所示。

表 5-1 市面上常见的一些 Sharding 中间件产品对比

功　能	Cobar	MyCat	TDDL	Shark
是否开源	开源	开源	部分开源	开源

续表

功　能	Cobar	MyCat	TDDL	Shark
架构模型	Proxy	Proxy	应用集成	应用集成
数据库支持	MySQL	任意	MySQL、Oracle	MySQL
外围依赖	无	无	Diamond	无
读写分离	支持	支持	支持	支持
分库分表	支持	支持	支持	支持

无论选择哪一款 Sharding 中间件产品，只要能够合理且高效地满足自身业务场景，那么它就是一款优秀的中间件产品。如果开发人员希望拥有大量的技术文档作为支撑，那么可以在生产环境中使用 Shark 作为分布式数据层来实现数据库分库分表后的数据路由操作，本书后续小节都会以 Shark 为例进行使用和配置讲解。

5.2.2 Shark 简介

Shark 是一款采用 Apache License 2.0 开源协议的分布式 MySQL 分库分表中间件。Sharding 领域的一站式解决方案，具备丰富、灵活的路由算法支持，能够方便 DBA 实现库和表的水平扩容，以及有效降低数据的迁移成本。Shark 站在巨人（SpringJDBC、Druid）的肩膀上，采用应用集成架构，放弃通用性，只为换取更好的执行性能，以及降低分布式环境下外围系统的宕机风险。

目前，Shark 每天为不同的企业、业务提供过亿级别的 SQL 读/写服务。Shark 的项目地址为 https://github.com/gaoxianglong/shark。Shark 的优点如下所示：

- 完善的技术文档支持；
- 动态数据源的无缝切换；
- 丰富、灵活的分布式路由算法支持；
- 非 Proxy 架构，应用直连数据库，降低外围系统依赖所带来的宕机风险；
- 业务零侵入，配置简单；

- 站在巨人的肩膀上（SpringJDBC、基于 Druid 的 Sqlparser 完成 SQL 解析），执行性能高效、稳定；
- 提供多机 SequenceID 的 API 支持，解决多机 SequenceID 难题；
- 默认支持基于 ZooKeeper、Redis3.x Cluster 作为集中式资源配置中心；
- 基于 Velocity 模板引擎渲染内容，支持 SQL 语句独立配置和动态拼接，与业务逻辑代码解耦；
- 提供内置验证页面，方便开发、测试及运维人员对执行后的 SQL 进行验证；
- 提供自动生成配置文件的 API 支持，降低配置信息出错率。

5.2.3 Shark 的架构模型

数据库实施分库分表后，就需要引入分布式数据层提供的数据路由支持来完成 SQL 语句的读/写操作。简而言之，就是以 Shard Key 为条件，按照特定的路由算法和规则对多数据源进行动态切换，这样即可定位到目标业务库上，然后再将 SQL 语句中的全局表名进行解析和重写替换（比如替换前：SELECT* FROM tab WHERE uid = 1，替换后：SELECT * FROM tab_0001 WHERE uid = 1），这样即可定位到目标业务表上，这就是 Sharding 中间件最核心的基础功能。

Shark 底层持有一组数据源集合，根据 SQL 语句中包含的 Shard Key 进行运算，然后再通过 Route 技术路由到具体的业务库和业务表上进行读/写操作。在此需要注意，Shark 内部并没有实现自己的 DBConnectionPool，这就意味着，开发人员可以随意切换任意的 DBConnectionPool 产品，如果觉得 C3P0 没有 BonePC 性能高，那么可以切换为 BonePC；如果觉得 BonePC 不如 Druid 稳定，又可以切换为 Druid。

开发人员完全不需要关心分库分表后的数据路由工作，在程序中就像是在操作单库和单表一样简单，并且执行性能更高效，Shark 就承担着这样的一个任务。Shark 的整体架构如图 5-5 所示。

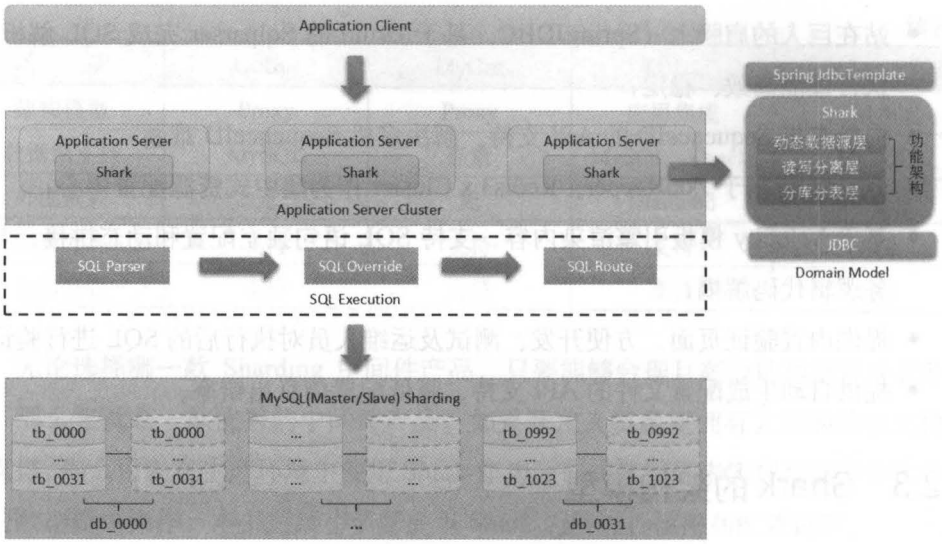


图 5-5 Shark 的整体架构

Shark 的领域模型位于 Spring 的 JdbcTemplate 和 JDBC 之间，也就是位于分布式数据层，Shark 站在巨人的肩膀上，这个巨人正是 Spring。简而言之，Shark 动态数据源层的 SharkDataSourceGroup 类派生自 Spring 的 AbstractRoutingDataSource 类，并通过 Spring AOP 的方式在运行时采用 Druid 的 Sqlparser 完成对 SQL 语句的解析工作，在此基础之上达到读/写分离、数据路由等操作目的，因此从另一个侧面反应出了 Shark 的源码注定是简单、健壮、易阅读和易维护的，因为 Shark 的核心功能只专注于 Sharding。反观一些其他的 Sharding 中间件内部实现，不仅需要实现 DBConnectionPool、动态数据源，还需要实现一些额外的功能，比如通用性支持、多种类型的 RDBMS 或 NoSQL 支持，势必会存在一些臃肿。Shark 的三层功能架构如图 5-6 所示。

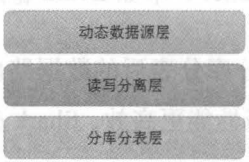


图 5-6 Shark 的三层功能架构

根据官方发布的 Shark Benchmark 结果来看，Shark 执行 SQL 语句时的效率是非常高效的，基本上损耗可以达到忽略不计的程度。关于更多测试结果，大家可以参考 https://github.com/gaoxianglong/shark/wiki/shark-benchmark-result。

5.2.4 使用 Shark 实现分库分表后的数据路由任务

目前 Shark 官方发布的最新版本为 2.0.2.RELEASE，开发人员可以通过 Maven 依赖的方式下载最新版本的 Shark 构件，如下所示：

```
<dependency>
    <groupId>com.sharksharding</groupId>
    <artifactId>shark</artifactId>
    <version>2.0.2.RELEASE</version>
</dependency>
```

Shark 项目的 pom.xml 文件中依赖有一些其他的第三方构件（依赖的 Spring 版本为 3.2.13.RELEASE），如果觉得版本过低或者与当前项目中依赖的构件产生冲突，可以在项目的 pom.xml 文件中使用 Maven 提供的<exclusions/>标签排除依赖，自行下载指定版本的相关构件即可。除此之外，大家还需要注意一点，Shark 的源码是在 Java 7 版本上进行编译的，因此在项目中使用 Shark 时，JDK 版本应该高于或等于 Java 7。

Shark 支持两类分库分表模式，如下所示：

- 单库多表模式；
- 多库多表模式。

单库多表模式应用在需要“垂直分库、水平分表”的场景下。前面曾经提及过，所谓垂直分库就是企业根据自身业务的垂直划分，将原本冗余在单库中的数据表拆分到不同的业务库中，实现分而治之的数据管理和读/写操作。水平分表是将原本冗

余在单库中的单个业务表拆分为 *n* 个“逻辑相关”的业务子表，不同的业务子表各自负责存储不同区间的数据，对外形成一个整体。假设垂直分库后的订单业务库的内部需要水平拆分出 1024 张订单表，那么这些订单子表就可以按照 order_0000 至 order_1023 的方式分布在订单业务库中。

和单库多表模式不同的是，多库多表模式应用在业务库和业务表都需要进行水平化的场景下，因此可以将其看作对单库多表模式的一种升级，同时这也是互联网场景下，关系型数据库应对高并发、单表数据量过大问题的最终解决方案。本书以多库多表模式为例，为各位读者演示如何在项目中配置和使用 Shark 进行分库分表后的数据路由。至于单库多表模式，大家可以参考 Shark 官方的用户指南进行配置和使用。

多库多表模式下 Shark 的主要配置信息如下所示：

```
<!-- 多库多表模式 -->
<bean id="shardRule" class="com.sharksharding.core.shard.ShardRule"
    init-method="init">
 <!-- 分库分表开关 -->
<property name="isShard" value="true" />
 <!-- 数据库读写分离 -->
    <property name="wr_index" value="r32w0" />
 <!-- 分库分表模式 -->
    <property name="shardMode" value="true" />
    <property name="consistent" value="true" />
 <!-- 分库路由算法 -->
    <property name="dbRuleArray" value="#key1|key2# % 1024 / 32" />
 <!-- 分表路由算法 -->
    <property name="tbRuleArray" value="#key1|key2# % 1024 % 32" />
 <!-- 表后缀拼接规则 -->
    <property name="tbSuffix" value="_0000" />
```

```
    </bean>
     <!-- 配置数据源 -->
    <bean  id="dataSourceGroup"  class="com.sharksharding.core.shard.
SharkDatasourceGroup">
        <property name="targetDataSources">
            <map key-type="java.lang.Integer">
                <entry key="0" value-ref="dataSource1" />
                <!-- 省略引用的数据源配置 -->
                <entry key="1023" value-ref="dataSource1024" />
            </map>
        </property>
    </bean>
```

由于笔者项目的特殊性，为了提升数据库的并行处理能力，以及避免单表数据量过大导致检索效率低下，我们不仅对单库进行了水平分库，还将单表进行了水平分表，最后一共拆分出了 1024 张业务子表，均匀分布在 32 个 Master 库中。也就是说，每个库中都包含 32 张业务表（1024/32=32），当然 Slave 库和 Master 库是保持一致的，如图 5-7 所示。

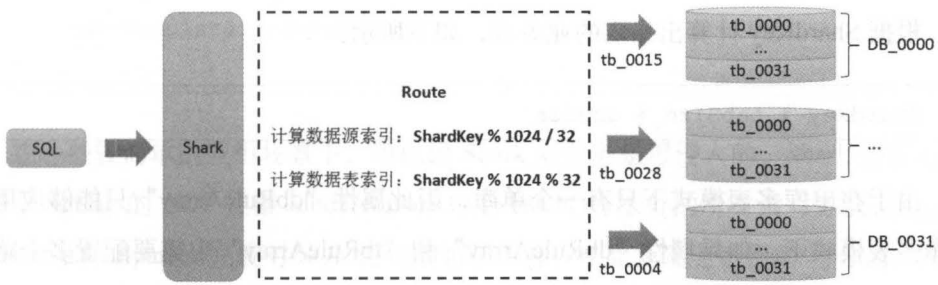


图 5-7　Shark 数据路由过程

在上述配置文件中，属性“isShard”定义了分库分表的开关，“true”为开启，默认为“false”，如果在程序中不开启分库分表功能，Shark 仍然可以为单库提供读写分离支持。

属性“wr_index”定义了数据库主从读写分离的起始索引，值“r32w0”指定了Master 的数据源索引范围在 0 到 31 之间，而 Slave 的数据源索引范围在 32 到 63 之间，当 Shark 解析 SQL 时，会根据操作类型自动判断和选择对应的主从数据源（如果不希望配置读写分离功能，可以将“wr_index”的值设置为“r0w0”）。

属性“shardMode”和“consistent”定义了 Shark 的路由模式，当“shardMode”的值被设定为“true”时，采用多库多表模式，反之采用单库多表模式；属性“consistent”只能够应用在多库多表模式下，当值被设定为“true”时，意味着表名后缀是连续的，简单来说，表名会按照 tab_0000 至 tab_1023 分布在所有的库中；将值设定为“false”时，每个库中的表名都统一按照 tab_0000 至 tab_0031（tbSize/dbSize=32）进行分布；表名后缀的拼接规则可以通过属性“tbSuffix”进行设定，默认为“_0000”。

属性“dbRuleArray”和“tbRuleArray”定义了 Shark 的路由算法。

根据 ShardKey 计算出指定的数据源，如下所示：

```
ShardKey % tabSize / dbSize
```

根据 ShardKey 计算出指定的业务表，如下所示：

```
ShardKey % tabSize % dbSize
```

由于在单库多表模式下只有一个单库，因此属性“dbRuleArray”只能够应用在多库多表模式下。如果属性“dbRuleArray”和“tbRuleArray”中需要配置多个路由条件时，可以通过符号“|”进行分隔。

Shark 规定了路由条件的数据类型必须是整数类型，如果是其他数据类型时（如 String 类型），可以先做一次 Hash 运算，为了避免 Hash 碰撞，可以带上 Hash 前的值（比如 SELECT * FROM tab WHERE email_hash = ? AND email = ?）。假设路由条件

的值为“123456”，那么根据 Shark 的路由算法进行运算后，得出的数据源索引为 18，表索引为 0，这也就意味着，执行 SQL 时 Shark 会路由到第 19 个库中的第 1 个表上进行读写操作。

当对单库进行水平分库时，建议大家遵循“二叉树分库”策略，即当 DBA 对数据库进行扩容的时候，都是以 2 的倍数进行扩容（比如从 2 台扩容到 4 台，从 4 台扩容到 8 台，从 8 台扩容到 16 台，以此类推），采用这种分库方式的最大优点就是扩容和数据迁移会更加方便。当对单表进行水平分表时，为了确保数据能够均匀落盘，建议大家将分表数量设定为分库数量的倍数（比如库的数量为 2，片的数量为 4；库的数量为 4，片的数量为 8；库的数量为 8，片的数量为 16，以此类推）。

成功配置好 Shark 的相关配置信息后，再来看看如何在程序中使用 Shark，如下所示：

```
@Resource
private JdbcTemplate jdbcTemplate;
public void testInsert() throws Exception {
    final String SQL= "INSERT INTO tab(c1,c2) VALUES(1,2)";
    jdbcTemplate.update(SQL);
}
```

在一些较简单的应用场景下，可以说 Shark 对业务是零侵入的，Shark 会在运行时基于 Spring 的 AOP 拦截 JdbcTemplate 中除 batch()方法外的所有读写方法，然后利用 Druid 的 Sqlparser 完成对 SQL 语句的解析工作，在此基础上达到读/写分离、数据路由等操作的目的。在此需要注意，Shark 并不支持 SpringJDBC 中除 JdbcTemplate 之外的 JDBC 模板类（比如 NamedParameterJdbcTemplate 和 SimpleJdbcTemplate），因为它们的底层实现都是通过调用 JdbcTemplate 的 API 来完成 SQL 语句的 CRUD 操作，并且从追求性能效率的角度来看，组件的封装层次越低执行性能越好。

或许有些人已经产生了疑问，在实际的开发过程中，我们往往不会将 SQL 语句耦合在业务代码中，而是将其配置在独立的 SQL 文件中，除此之外，支持 SQL 语句的动态拼接功能似乎也是刚需。尽管 SpringJDBC 默认并没有提供这样的功能，但值得庆幸的是，Shark 很好地填补了这块领域的空白。SQL 语句独立配置，如下所示：

```
<bean id="sqlTemplate" class="com.sharksharding.sql.SQLTemplate">
    <constructor-arg name="path" value="classpath:sql.xml" />
</bean>
```

Shark 支持从 classpath 目录下加载独立配置的 SQL 文件，也支持直接从绝对路径上进行加载。从 SQL 文件中获取出指定的 SQL 语句，如下所示：

```
@Resource
private SQLTemplate sqlTemplate;
@Resource
private JdbcTemplate jdbcTemplate;
@Override
public List<TabInfo> testQuery(Map<String, Object> params)
        throws Exception {
 /* 从 SQL 文件中获取出指定的 SQL 语句 */
final String SQL = sqlTemplate.getSql("key", params);
return jdbcTemplate.query(SQL, tabMapper);
}
```

在上述程序中，通过 SQLTemplate 提供的 getSql(String key, Map<String, ?> params)方法即可成功从 SQL 文件中获取出指定的 SQL 语句。其中，第 1 个参数为定义在 SQL 文件中的 SQL 名称，而参数 params 则为需要传递给 SQL 语句的一系列参数集合。

再来看看 SQL 语句应该如何进行定义，Shark 支持 XML 和 Properties 两种文件形式来定义 SQL 语句，本书仅以 XML 文件格式为例，如下所示：

```
<sqls>
   <sql name="queryTab">
      <![CDATA[
         SELECT * FROM tab WHERE c1=${c1} AND c2='${c2}'
      ]]>
</sql>
</sqls>
```

根标签为<sqls/>，每一个 SQL 语句都包含在一个独立的<sql/>标签中。其中，属性 name 用于定义 SQL 名称，不允许重复出现。由于 Shark 是基于 Velocity 模板引擎渲染内容，因此也支持 Velocity 语法来实现 SQL 语句的动态拼接，如下所示：

```
<sql name="queryTab">
   <![CDATA[
      SELECT * FROM Tab
      #if(${c1})
         WHERE c1 = ${c1}
      #end
      #if(${c2})
         AND c2 = '${c2}'
      #end
   ]]>
</sql>
```

对于开发人员而言，使用 Shark 只需要掌握 SpringJDBC 的基本用法即可，不会再有额外的学习成本，因此，开发人员无须关心数据库分库分表后的数据路由工作，使用 Shark 就像在操作单库和单表一样简单。关于 Shark 的更多使用方式，本

书不再一一进行讲解，大家可以参考 Shark 的用户指南。

最后不得不提醒大家，如果 SQL 语句中的第一个参数并不是定义在配置信息中的路由条件时，那么 Shark 将抛出如下异常信息：

```
com.sharksharding.exception.SqlParserException: can not find shardkey
```

5.2.5 分库分表后所带来的影响

一旦数据库实施分库分表后，多多少少都会对开发人员产生一定的影响，而这类影响大多体现在逻辑代码的实现上。假设我们从单库单表的数据库架构演变为“垂直分库、水平分表”或者“水平分库、水平分表”架构时，有四个比较棘手的问题是需要慎重考虑和尽早规划的，如下所示：

- ACID [Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）、Durability（持久性）] 如何保证；
- 多表之间的关联查询如何进行；
- 无法继续使用外键约束；
- 无法继续使用由 Oracle 提供的 Sequence，或者 MySQL 提供的 AUTO_INCREMENT 生成全局唯一和连续性 ID。

关于 SequenceID 应该如何生成，大家可以直接阅读 5.2.6 节；关于分布式事务，本书在后续都有着重讲解。

任何大型的互联网企业，一旦数据库实施分库分表后，对于 SQL 语句的编写都有一条铁打不动的“军规”，那就是更倾向于简单化、轻量化。简而言之，就是尽可能地避免使用多表联合查询语句，而是将其拆分为多条单表查询语句，多次查询。或许有人会产生疑问，难道多表联合查询性能不如多次单表查询快吗？其实不然，

至少从测试结果来看，多次单表查询在执行效率上并不比多表联合查询慢多少，并且在互联网场景下，为了扩展性、易读性和维护性，牺牲一点性能（如多次会话连接、资源消耗），so what?

在互联网场景下，利用单表查询语句换来的优势，主要有以下两点：

- 查询语句简单，易于理解、维护和扩展；
- 缓存利用率高。

5.2.6 多机 SequenceID 解决方案

在面向单点系统的场景下，生成一个全局唯一的 ID 是非常简单的事情。以数据库为例，Oracle 提供的 Sequence、MySQL 提供的 AUTO_INCREMENT 都可以生成不重复且连续的 ID，甚至开发人员还可以在程序中使用 UUID.randomUUID()方法进行唯一 ID 的生成。总的来说，这类生成唯一 ID 的方式都具备一个共同的特点——面向单点。

但是数据库实施分库分表后，将要面临的场景就不再是单点环境了，而是分布式环境，因此如何有效地生成 ID 自然也就成为了一个令人头痛的问题。首先如何保证所生成的 ID 具备唯一性是重点，其次如果需要考虑连续性，那么将更加复杂，因为这里所指的连续性不再是由单点维护，而是整个分布式环境下的连续性。当然，如果只是考虑生成 ID 的唯一性而忽略掉连续性，开发人员可以利用 UUID、物理机器 IP、随机值、时间戳等不同的维度共同生成唯一的 ID。既然要兼顾生成 ID 的唯一性和连续性，那么依赖一个独立的外围单点系统来负责完成则不失为一个可取的方案。

值得庆幸的是，Shark 内部已经提供了生成 SequenceID 的 API（底层支持数据库和 ZooKeeper 作为申请 SequenceID 的存储系统），开发人员只需要调用即可。接下来

笔者就以MySQL数据库为例，为大家演示应该如何使用Shark提供的API获取兼顾唯一性和连续性的SequenceID。

使用Shark提供的DDL语句在单库中创建所需的数据表，如下所示：

```
CREATE TABLE shark_sequenceid(
    s_id INT NOT NULL AUTO_INCREMENT COMMENT '主键',
    s_type INT NOT NULL COMMENT '类型',
    s_useData BIGINT NOT NULL COMMENT '申请占位数量',
    PRIMARY KEY (s_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_bin;
```

准备工作完成以后，通过以下方式调用即可：

```
final static String NAME = "root";
final static String PWD = "123456";
final static String URL = "jdbc:mysql://ip:port/db";
final static String DRIVER = "com.mysql.jdbc.Driver";
public @BeforeClass static void init() {
    /* 初始化数据源信息 */
    SequenceIDManger.init(NAME, PWD, URL, DRIVER);
}
public @Test void getSequenceId() {
    /* 获取 SequenceId */
    System.out.println(SequenceIDManger.getSequenceId(100, 10, 5000));
}
```

除了可以直接使用JDBC的方式与数据库建立会话获取SequenceID，在实际的开发过程中，开发人员更倾向于使用数据库连接池的方式与数据库打交道，此时Shark的ID生成器也支持配置数据库连接池，如下所示：

```
private DruidDataSource dataSource;
public @BeforeClass void init() {
    dataSource = new DruidDataSource();
/* 省略数据源信息配置相关代码 */
    SequenceIDManger.init(dataSource);
}
public @Test void getSequenceId(
/* 获取 SequenceId */
    System.out.println(SequenceIDManger.getSequenceId(100, 10, 5000));
}
```

Shark 提供生成 SequenceID 的 API 只是封装了 ID 的生成逻辑，真正保证唯一性和连续性的还是单点数据库（想要提升容灾能力和 HA，可搭建 Master/Slave 模式）。上述程序中的 getSequenceId(int idcNum,int type,long memData）方法包含 3 个参数，第 1 个参数是 IDC 机房编码，用于区分不同的 IDC 机房；第 2 个参数是业务类别，2 位数字长度；第 3 个参数是向数据库申请的 ID 缓存数。最终这段代码会返回一个数据类型为 long、数值长度为 19 位的 SequenceID。数值长度为 19 位的 SequenceID 已经能够满足绝大多数企业后续多年的使用了。

或许当大家看到这里时，会有一头雾水的感觉，接下来笔者就为大家阐述 Shark 生成 SequenceID 的原理。如果每一个应用系统中都依赖 Shark，也就意味着每一个应用系统都间接集成一个 ID 生成器，并统一指定一个单点存储系统作为存储介质，如图 5-8 所示。数据库主要负责存储当前 ID 序列的最大值，每次每个 ID 生成器从数据库中申请 ID 时，都会通过行锁机制来确保并发环境下数据的一致性。或许大家会产生一个疑问，是否存在性能瓶颈？如果每生成一个 ID 都去数据库申请必然性能低下，但是 Shark 的 ID 生成器一次会从数据库中取出一段 ID，然后缓存在本地，这样就不用每次都去数据库中申请了。

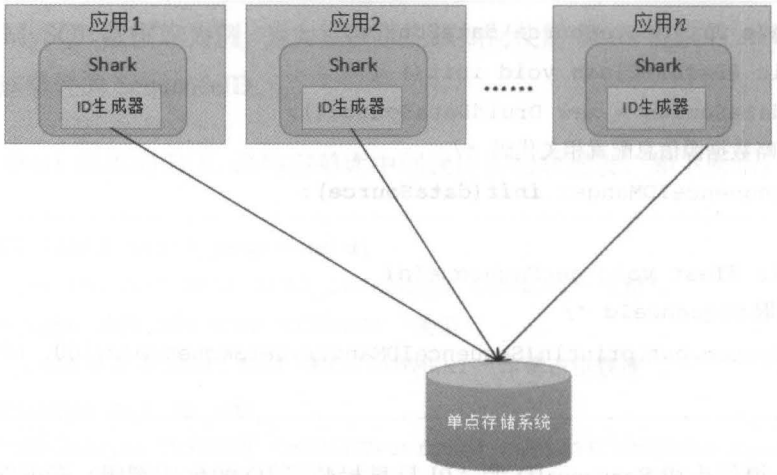


图 5-8 Shark 的 ID 生成器

想要提升 Shark 生成 SequenceID 的性能，可以调整 getSequenceId()方法中的第 3 个参数，该数值越大 ID 生成效率越高，但是一旦 ID 生成器取了一段 ID 后，系统突然宕机了，那么这一组 ID 将会废弃，并且后续可向数据库申请的有限 ID 缓存数将越来越少（SequenceID 的最大值不可超过二进制位数 64 位 long 类型的最大值 9223372036854775807），因此这需要大家仔细权衡，笔者建议将其值设置在“5000～10000”较为合理。

5.2.7 使用 Solr 满足多维度的复杂条件查询

Solr 是 Apache 旗下的一个子项目，它是一种采用 Java 语言编写、开放源代码，并且扩展自 Lucene 的搜索引擎。尽管 Solr 和 Lucene 同属 Apache，但是两者并不存在直接的竞争关系，相反的，Solr 非常依赖于 Lucene，因为 Solr 的核心功能正是通过调用 Lucene 的 API 来实现的。

Solr 能够非常方便地在 Web 容器中进行部署，并且它提供了一系列的 HTTP 接口供开发人员调用，使之向服务器提交 Document，对索引进行创建、修改和删除等

操作。除此之外，Solr 还提供了一整套完整的表达式语言，能够实现任意复杂的条件查询。Solr 的整体架构如图 5-9 所示。

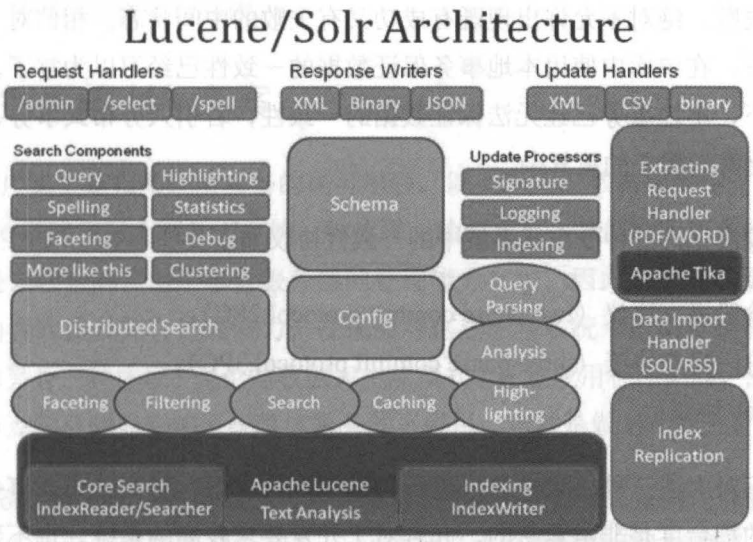


图 5-9 Solr 的整体架构

一般，在互联网场景下使用 Solr 有两个最主要的场景。

首先，如果数据库（MySQL）的查询条件采用 like 进行模糊查询（比如 SELECT * FROM item WHERE item_name like '%香水%'），数据库是不会进行索引的，而是采用全表扫描机制。因此当表中数据量较大时，查询速度会变得异常缓慢，就像我们在购物网站搜索某一类商品时，由于物品种类成千上万，使用 like 进行模糊查询就会非常拙劣，因此在这种场景下使用 Solr 将非常适合。

其次，数据库实施分库分表后，数据最初以什么样的维度进行落盘，最后就只能够通过这种维度进行查询，如果希望同时满足多维度的复杂条件查询需求，利用 Solr 这类搜索引擎将非常容易。

5.2.8　关于分布式事务

所谓事务，指的是程序中多个操作对上下文的影响是一致的，要么同时成功，要么一起失败，绝对不允许出现既有成功又有失败的中间状态。相信对于大多数开发人员而言，在单库中使用本地事务保证数据的一致性已经习以为常了。但是在分布式环境下，本地事务已经无法保证数据的一致性，若引入分布式事务，所带来的问题往往比我们想象的更复杂。

目前较常见的三种分布式系统中的一致性协议如下：

- 两阶段提交协议（two phase commit protocol,2PC）；
- 三阶段提交协议（three phase commit protocol,3PC）；
- Paxos 协议。

本书不对上述三种一致性协议进行过于深入的讲解，毕竟分布式事务的实施与业务本身的耦合度是非常紧密的，并且对于并发要求较高的系统，也不建议引入分布式事务，最主要的原因是存在性能瓶颈问题。

以两阶段提交协议为例，首先提交操作会涉及多次节点之间的网络通信；其次由于事务时间、资源锁定时间延长，会导致资源等待时间变长。因此在某些场景下，能够不引入分布式事务的还是尽量不要引入，有些无关紧要的数据丢失就丢失了，可以无须理会，但是一些比较重要的数据，如果一定需要保证一致性，也不要刻意去追求强一致性，可以考虑采用基于消息中间件保证数据最终一致性方案，让之前执行失败的操作继续向前执行下去。

举个生活中的小例子，比如我们与家人去餐厅吃饭，如果遇上周末或节假日，那么门口一定是人山人海，相信这种场景大家一定都非常有感触，那么餐厅为了提升自己的接待能力，会为排队就餐的那些人发放一张等待叫号的排队码，等有空位的时候服务员便会通知人们进去用餐。当然，在这段时间内，我们可以自行选择去逛逛商场。手中的这个排队码就可以理解为消息队列中的一条消息（凭证），只要不丢失，

最终我们还是可以顺利进入餐厅用餐的，这就是生活中常见的最终一致性案例。

本书在 5.4 节演示了如何通过使用消息中间件的方式保证数据最终一致性的实战案例，大家可以直接进行阅读。

5.3 数据库的 HA 方案

HA 在广义上是指系统所具备的高可用性。以 MySQL 数据库为例，DBA 实施 Master/Slave 搭建实际上就是 HA 的一种体现。除此之外，我们的应用系统在大部分情况下都是无状态的，由于单台服务器的处理能力有限，因此为了能够更好地提升系统整体的吞吐量及并行处理能力，往往会对这些应用系统采用集群部署，这也是实施 HA 的另外一种体现。无论是数据库主从模式，还是应用程序集群，往往都不会因为单一节点的故障而影响系统整体服务的不可用，这就是做 HA 的主要原因。

如果 Tomcat 这类 Web 容器需要部署集群，前端还需要引入反向代理服务器来分发请求，在互联网领域使用 Nginx+Tomcat 这种形式的集群模式非常常见，由 Nginx 负责请求分发、负载均衡，以及 Failover 等任务。数据库如果需要搭建 HA，也需要一种机制保证主从的正常切换，保证 Master 宕机后 Slave 能够开放读写权限，充当新 Master 的角色，如图 5-10 所示。目前有三种成熟的方案可供选择，如下所示：

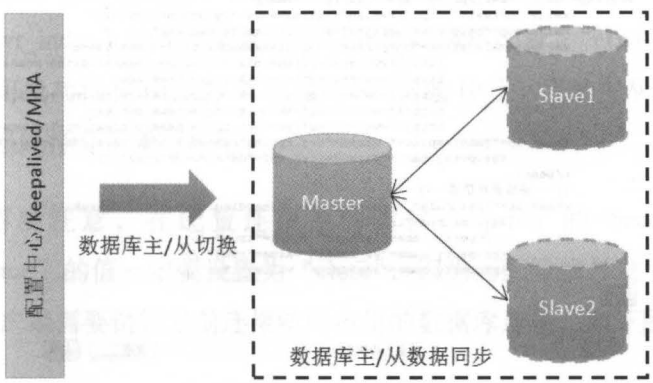


图 5-10 MySQL 数据库 HA 架构

- 基于配置中心实现主从切换；
- 基于 Keepalived 实现主从切换；
- 基于 MHA 实现主从切换。

5.3.1 基于配置中心实现主从切换

数据库实现主从切换通常有两种形式比较常见，一种是通过监控系统发出告警后，运维人员手动修改数据源信息，另一种则是 Master 实例发生故障后自动切换到 Slave 库上。在开始讲解本节的内容之前，如果你对配置中心的作用还不是很了解，那么笔者建议你先熟悉一下第 3 章中的相关内容，然后再来阅读本节。

如果采用手动切换主从的方式实现数据库的 HA，那么基于配置中心是一个非常不错的选择，但是这往往需要配置中心客户端的支持才能够实现，如图 5-11 所示。值得庆幸的是，Shark 目前已经提供了基于 ZooKeeper 的配置中心客户端，方便开发人员将数据源信息统一配置在 ZooKeeper 中。

图 5-11 在配置中心配置数据源信息

将本地配置文件中的数据源配置信息修改为以下配置：

```
<bean class="com.sharksharding.resources.register.bean.RegisterDataSource"/>
<bean class="com.sharksharding.resources.conn.ZookeeperConnectionManager"
    init-method="init">
    <constructor-arg index="0" value="${address}" />
    <constructor-arg index="1" value="${session.timeout}" />
    <constructor-arg index="2" value="${nodepath}" />
</bean>
```

通过上述配置，当 Spring 的 IOC 容器启动时，便会与 ZooKeeper 建立会话连接，并将数据源信息加载到本地。由于使用了 Shark 的配置中心客户端，程序中基于注解自动装配的 JdbcTemplate 需要调整为以下形式：

```
public void testInsert() throws Exception {
    JdbcTemplate jdbcTemlate = GetJdbcTemplate.getJdbcTemplate();
    final String SQL= "INSERT INTO tab(c1,c2) VALUES(1,2)";
jdbcTemplate.update(SQL);
}
```

将数据源信息配置在配置中心后，一旦 Master 实例发生故障，我们只需要修改 Shark 属性“wr_index”的主从起始索引（比如故障前配置：“r1w0”，修改后配置：“r1w1”），即可成功将 Master 的 IP 切换到 Slave 的 IP 上，当然主从切换之前我们必须确保 Slave 已经打开了读写权限。

最后还需要注意，在配置连接池的时候，Spring 的<bean/>标签中属性“destroy-method”的值一定要设置为“close”，因为当配置信息发生变更时，Shark 的配置中心客户端需要销毁之前连接池所占用的数据库连接，如下所示：

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
    init-method="init" destroy-method="close">
```

5.3.2 基于 Keepalived 实现主从切换

当 Master 实例发生故障时，监控系统会发出告警通知，这时运维人员可以通过修改配置中心配置的数据源信息来达到手动切换数据库主从的目的。当然，除了采用这种方式实现数据库的 HA，还可以通过自动主从切换的方式实现数据库的 HA。

本书以 Keepalived 为例，假设 Master 机器和 Slave 机器上都装有 Keepalived 程序，那么我们需要先修改/etc/keepalive 目录下的 keepalive.conf 配置文件，在主从机器上都配置好 VIP（Virtual IP Address，虚拟 IP 地址），如下所示：

```
# 在Master机器（192.168.1.20）和Slave机器（192.168.1.30）上同时指定的VIP地址
virtual_ipaddress {
192.168.1.10
}
# Master机器上的Keepalived主要配置
virtual_server 192.168.1.10 3306 {
    delay_loop 2
    lb_algo wrr
    lb_kind DR
    persistence_timeout 60
    protocol TCP
    real_server 192.168.1.20 3306 {
        weight 3
        #检测到数据库实例宕机后可执行的脚本
        notify_down /root/test.sh
        TCP_CHECK {
            connect_timeout 10
            nb_get_retry 3
```

```
            delay_before_retry 3
            connect_port 3306
        }
    }
    }
    # Slave 机器上的 Keepalived 主要配置
    virtual_server 192.168.1.10 3306 {
        delay_loop 2
        lb_algo wrr
        lb_kind DR
        persistence_timeout 60
        protocol TCP
        real_server 192.168.1.30 3306 {
            weight 3
            notify_down /root/test.sh
            TCP_CHECK {
                connect_timeout 10
                nb_get_retry 3
                delay_before_retry 3
                connect_port 3306
        }
    }
    }
```

成功启动 Master 机器和 Slave 机器上的数据库和 Keepalived 后，程序中数据源的配置和之前相比有些区别，Slave 库的 IP 指向的是 Slave 机器的物理 IP，而 Master 库的 IP 指向的则是 VIP 地址。

在程序运行的过程中，Master 机器和 Slave 机器上的 Keepalived 程序会相互发送心跳信号，确认对方状态是否存活，一旦 Master 实例出现故障，根据所配置属性“delay_loop”的值，Keepalived 会在指定的时间范围内定时检查数据库的健康状态，

一旦发现异常，Master 机器上的 Keepalived 会选择“自杀”，这时 Slave 机器上的 Keepalived 由于检测不到心跳，便会开始接管 VIP 请求，这样所有的写入请求就会全部落到 Slave 库上，如图 5-12 所示。

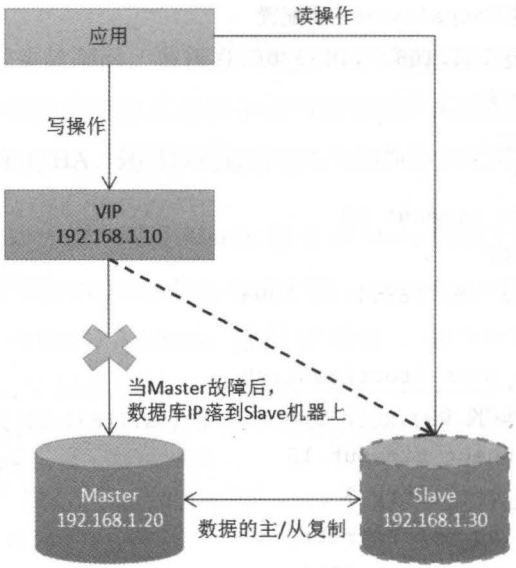


图 5-12 基于 Keepalived 实现数据库 HA

我们可以在 Keepalived 的配置文件中通过属性“notify_down”来指定当检测到数据库实例宕机后需要执行的脚本（比如将 Slave 库的读写权限打开）。在此需要注意，从 Master 实例宕机到 Slave 接管成为新 Master 之前的这段时间，数据库暂时不可用，如果这时有写入请求进来，有可能造成数据丢失，那么怎样才可以避免在数据库主从切换的过程中不出现数据丢失呢？

在大部分情况下，为了缓解数据库的查询压力，数据都是优先插入到缓存服务器中再插入到数据库的。那么我们可以通过指定 Job 程序定时增量检查数据库与缓存数据是否一致，如果不一致，则可以利用缓存数据对数据库实施补偿。而且，在某

些场景下，为了消峰我们采用异步模型将数据插入到数据库，比如先将数据写入到消息队列，然后消费端消费到消息后再写入到数据库。因此，当消费消息后插入数据到数据库失败时，可以通过 failover 机制尝试多次等一系列手段来保证数据库主从切换过程中数据尽可能不丢失。

5.3.3 保障主从切换过程中的数据一致性

在数据库主从切换的过程中，我们可以利用一系列的辅助手段来避免在数据库不可用时造成数据的丢失，但是在数据库主从切换过程中，还存在一个棘手的问题，那就是 Master 实例宕机后，Slave 变为新的 Master 时，主从数据肯定不一致了，此时如何保证主从数据的一致性呢？

在解决这个问题之前，我们首先回到主从同步这个问题上，在 Master 的 TPS 较高的情况下，主从同步的延迟肯定是非常大的，因此为了避免数据库读写分离后应用层无法从 Slave 拉到实时数据，通常的做法是在写入 Master 之前也将同一份数据落到缓存中，以避免高并发情况下，从 Slave 中获取不到指定数据的情况发生。

当然，MySQL 本身提供的机制也能够在最大程度上保证主从数据的一致性，参考微信抢红包的案例，在非高峰期或者非活动日时，由于流量不会特别大，因此数据库主从同步之间可以显式开启半同步复制（Semi-synchronous Replication）功能，如图 5-13 所示。这是 MySQL 在 5.5 版本时开始提供的一个功能，半同步复制可以理解为主从之间的强制数据同步，以保证主从数据的准实时性。简单来说，当事务提交到 Master 后，Master 会等待 Slave 的回应，待 Slave 回应收到 Binlog（二进制日志）后，Master 才会响应请求方已经完成了事务。在峰值流量较大的场景下，笔者不建议开启这项功能，这会对 TPS 产生一定的影响。

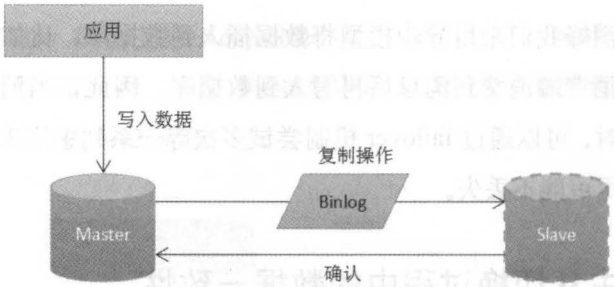


图 5-13 MySQL 的 Semi-synchronous Replication 机制

尽管半同步复制机制可以在主从数据库不宕机的情况下保证数据的一致性，但是当 Master 实例宕机后，Slave 摇身一变成为新的 Master 时，主从数据库之间的数据肯定会出现不一致。在这种情况下，为了避免手动比对 Binlog 来确保主从数据的一致性，可以使用 MySQL 在 5.6 版本开始提供的 GTID（全局事务 ID）特性。由于新 Master 是之前的 Slave，而宕机后的 Master 在重启后可以作为 Slave 存在，可以依靠 GTID 特性来保证主从之间数据的最终一致性。

5.4 订单业务冗余表需求

在互联网场景下，某些业务除了并发访问度较高，通常还伴随着数据量非常大的特点。一旦数据库实施分库分表后，订单业务的查询需求就会开始变得复杂。笔者曾经提及过，我们的数据最初以什么样的维度进行落盘，最后就只能够通过这样的维度进行查询。订单数据需要对此进行查询的往往同时包含卖家和买家两类用户群体，卖家需要查询哪些买家对自己店铺的商品进行了下单，而买家也需要通过订单信息完成后续的付款等操作，这就产生了矛盾，我们在处理订单数据时，到底是以 seller_id 为维度进行数据落盘，还是以 buyer_id 为维度进行数据落盘呢？

如果只是以其中一个维度进行数据落盘，那么最终能够查询出订单数据的只是卖家或买家中的一方。因此，为了应对这种特殊的业务需求，目前业界比较常见的

做法是对同一份订单数据进行冗余存储，即我们需要维护两张订单表，一张是卖家订单表，另外一张是买家订单表，如下所示：

```
#卖家订单表
t_order_seller(seller_id,buyer_id,order_id)
#买家订单表
t_order_buyer(buyer_id,seller_id,order_id)
```

卖家订单表以 seller_id 为维度进行数据落盘，买家订单表则以 buyer_id 为维度进行数据落盘，这样既可以满足卖家的查询需求，也能够满足买家的查询需求，这就是数据库实施分库分表后订单业务场景下常见的冗余表需求。

5.4.1 冗余表的实现方案

既然采用数据冗余方案可以解决和满足订单业务分库分表后不同维度的查询需求，接下来就为大家讲解冗余表需求的具体实现方案。实现订单数据的冗余存储，就是同时向 t_order_seller 表和 t_order_buyer 表中插入同一份订单数据。数据的写入过程，可以分为以下两种形式：

- 数据同步写入；
- 数据异步写入。

业务上实现订单数据的落盘，通常的做法是由接入层调用订单服务，然后由订单服务将订单数据写入订单表中，这是非冗余存储时数据落盘的实现方式。但是实现冗余表后，同一份数据需要写入两次，如果采用数据同步写入方案，订单服务会按照先后顺序，写完第一张表后再将数据同步写入到另外一张表中，如图 5-14 所示。实施这种方案基本不具备复杂度，就是由之前的单写变为双写。其缺点也非常明显，就是写入时间比之前增加了 1 倍，因此如果对系统的 TPS 有严格的时间要求，可以使用数据异步写入方案。

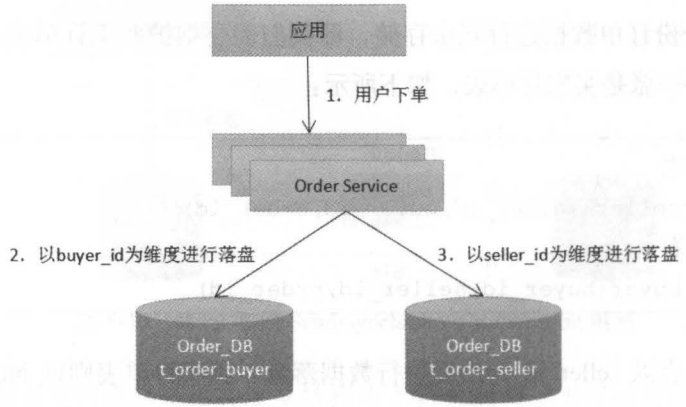


图 5-14　数据同步写入

采用数据异步写入方案后，订单服务不再采用同步双写落盘数据，而是当服务成功将数据写入到第一张表后，通过异步模型将数据写入到第二张表中（比如启动一个异步线程负责写入，或者写入消息队列后由消费者负责写入），如图 5-15 所示。由于第二次数据写入操作是异步的，服务不用等待其结果返回，系统整体性能会得到提升，但是相对于数据同步写入方案来说，系统复杂度却增加了。

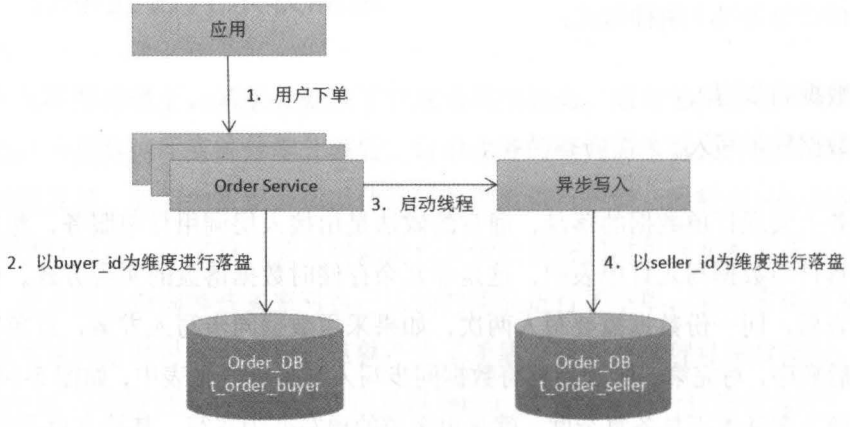


图 5-15　数据异步写入

在此需要注意，由于没有可靠的事务机制来保证数据双写过程中的一致性，因此无论是采用同步写入还是异步写入方案，在数据库 TPS 较高的情况下，冗余表极有可能出现数据不一致。因此这里就涉及一个问题，到底是优先写入到 t_order_seller 表更好，还是优先写入到 t_order_buyer 表更合适？

结合笔者项目中实际的订单业务来分析，优先将数据写入到 t_order_buyer 表中会更好，因为哪怕后续 t_order_seller 表写入数据失败，至少用户还可以继续推动订单状态的流转（如对订单完成支付），反之如果用户看不到系统新生成的订单，那么对于商家来说似乎也是没有任何意义的。

5.4.2 保障冗余表的数据一致性

笔者曾经提及过，数据库实施分库分表后对现有环境会产生一定的影响，数据的一致性难以保证，对于那些强调数据一致性的场景，我们能够做的要么是使用分布式事务，要么是采用最终一致性方案。但是分布式事务具有复杂性和低效性，因此采用最终一致性方案就显得顺理成章了。

在冗余表需求中，基于数据同步写入和数据异步写入两种方案，都可能在数据写入时产生失败，从而导致 t_order_seller 表和 t_order_buyer 表之间的数据产生不一致的情况发生，那么最终一致性方案应该如何实施呢？简单来说，当订单数据成功写入 t_order_buyer 表后，立即将消息写入到消息队列中，成功写入 t_order_seller 表后，再把相同的消息也写入到消息队列中，消费者消费到第 1 条消息后，如果在指定的时间范围内没有消费到第 2 条消息，就可以认为数据已经产生了不一致，需要执行数据补偿操作，这种补偿方案称为“线上检测补偿”，如图 5-16 所示。尽管采用“线上检测补偿”会让数据不一致的窗口期缩短，但是实现难度却比较复杂。

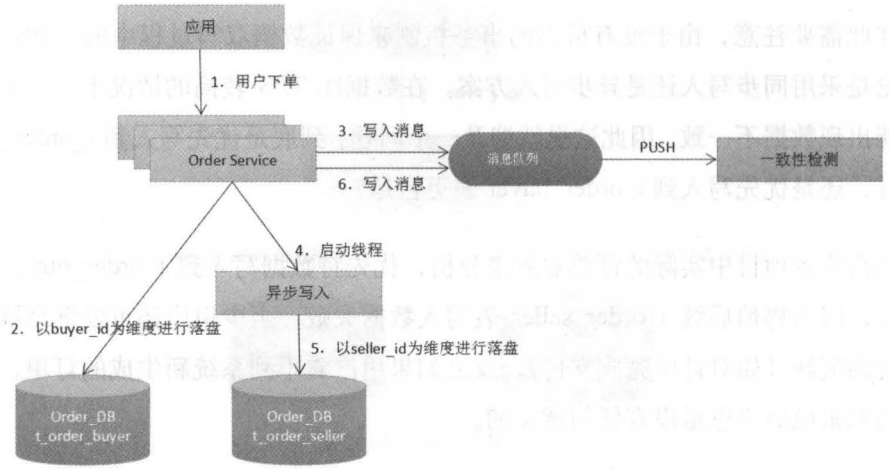


图 5-16 线上检测补偿

细化到具体的逻辑实现上，在消费者代码中我们可以申明两个 List 集合，分别用于存储接收到的 buyer 和 seller 的对等消息，如下所示：

```
/* 存储买家消息 */
Vector<String> buyerData = new Vector<String>()
/* 存储卖家消息 */
Vector<String> sellerData = new Vector<String>();
```

由于存储到 buyerData 和 sellerData 中的消息是一样的，那么假设制定 t_order_seller 表和 t_order_buyer 表之间数据不一致的窗口期为两秒，一旦 buyerData 或 sellerData 中任意一个不包含指定的消息时，就可以认为数据需要进行补偿。在这里存在一种特殊情况，假设消费者消费到第 1 条消息后，由于网络原因导致第 2 条消息消费延迟，但数据库之前已经成功写入，因此数据补偿之前，需要优先执行幂等操作。数据一致性检测与补偿的伪代码如下所示：

```
public void checkTest() {
    try {
```

```
            if (buyerData 中不包含指定的消息) {
                /* 从数据库中获取买家和卖家的订单数据 */
                if(数据库中不存在买家的订单数据){
                    /* 由 t_order_seller 补偿 t_order_buyer 数据 */
                }
            } else if (sellerData 中不包含指定的消息) {
                /* 从数据库中获取买家和卖家的订单信息 */
                if(数据库中不存在卖家的订单数据){
                    /* 由 t_order_buyer 补偿 t_order_seller 数据 */
                }
            } else {
                logger.debug("不需要进行数据补偿);
            }
        } catch (Exception e) {
            logger.error("数据补偿失败", e);
        } finally {
            /* 根据 key 删除 buyerData 和 sellerData 中的对等消息 */
        }
    }
```

如果你觉得“线上检测补偿”方案实施比较复杂，并且还需要部署其他外围系统来协作，容易增加系统整体的宕机风险，笔者还提供了另外一种补偿方案，那就是基于增量日志扫描的线下检测，如图 5-17 所示。

当订单数据成功写入 t_order_buyer 表后，立即将数据写入到 log1 中，成功写入 t_order_seller 表后，再把相同的数据写入到 log2 中，通过指定一个 Job 程序不停地增量比对 log1 和 log2，如果出现数据不一致的情况，则进行数据补偿。

相对于“线上检测补偿”方案来说，“线下检测补偿”方案更易实现，但是数据不一致的窗口期相对也会更长，至于程序中到底应该使用哪一种数据补偿方案，则需要结合实际的业务场景而定。

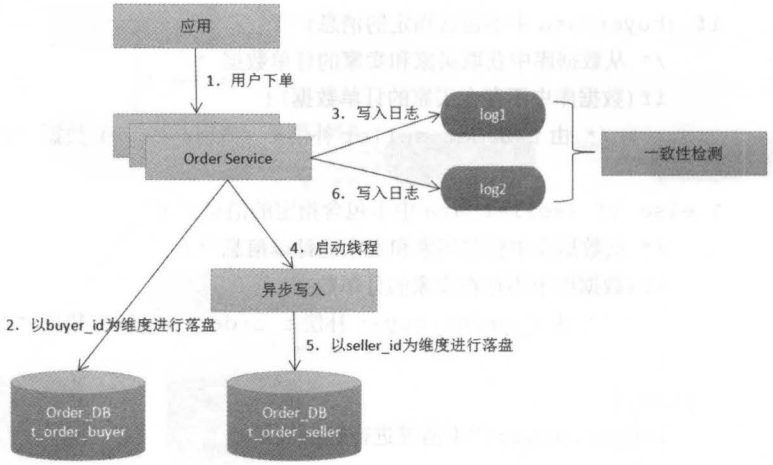


图 5-17　线下检测补偿

5.5　本章小结

笔者从关系型数据库的架构演变开始讲起，详细地为大家分析了在互联网场景下，关系型数据库中应对高并发、单表数据量过大的问题的最终解决方案。当大家弄清楚为什么关系型数据库需要进行分库分表后，笔者又实战演示了如何使用 Shark 中间件完成数据路由，以及分库分表后的诸多注意事项、数据库的 HA 方案等。笔者还根据实际的业务场景讲解了如何保障订单冗余表的数据一致性。

笔者在此留给大家一个值得深思的问题，随着市面上各种 NoSQL、NewSQL 的异军突起，传统 RDBMS 的地位正在一步一步地遭受着前所未有的威胁和冲击。由于使用关系型数据库的本质是看重其 ACID 等特性，但是随着并发、数据量的增大，使用 Sharding 似乎是最终手段，但由于 ACID、关联关系，以及外键约束等诸多特性相继被束缚，抛开 RDBMS 的稳定性和成熟度，以及开发人员根深蒂固的传统思维，在特殊的业务场景下是否可以将数据过渡到 NoSQL 中？

后记

阅读完本书，并不代表你就能够在今后的职业生涯中一帆风顺，甚至成为一名优秀的架构师或技术专家，因为在大型网站架构演变过程中等待我们解决的技术难题还有很多。大家千万不要被书名所迷惑，“人人都是架构师”，这本身其实只是一个噱头，笔者只是想用一种具有亲和力的方式让你放下对技术难题的恐惧，勇于克服它们，这样你才能够走得更远。

架构师到底是什么？它真的是开发人员口中所述的不会写代码的“技术大忽悠”吗？其实笔者也发现，目前互联网企业的架构师，真正奋斗在一线和程序员们一起Coding的确实太少。当然，并不是说Coding应该是架构师的主要工作，因为大企业的架构师们更主要的工作是负责把控方向，掌舵大局。笔者认为，作为一名合格的架构师，肯定是需要权衡一些东西的，哪个是主要矛盾就优先解决哪一个。但这并不代表给了你不写任何代码的权利，而是你应该比其他人做得更多，付出得更多，这样才能够使你的架构更接地气，更好地将架构落地。除此之外，架构师对于业务的理解一定要透彻，否则你的架构又为谁而设计呢？

如图 A-1 所示，一个优秀的架构师，抽象思维能力是必不可少的，架构师要善于“庖丁解牛”，将实物概念化并归类，比如一个大型网站，你能够迅速根据业务功能的不同，将业务垂直化；而扎实的技术功底又是架构师能力版图中所占比例最大

的一块，因为抽象思维能力是虚的，技术能力是实的，只有做到虚实结合，才能够达到“手中无剑，心中有剑”的境界；技术前瞻性是需要架构师凭借自身经验和直觉预估当前架构的缺陷会为将来埋下哪些隐患、哪些技术问题是需要在网站发展到一定阶段就必须重构的、哪些技术在未来是趋势，需要提前进行了解和学习的；多领域知识既要求了架构师的知识广度，又要求了架构师的知识深度，因为架构师的技术能力不能够仅局限在自己所擅长的那一亩三分地；沟通交流能力其实非常重要，因为大多数情况下，我们都是在与人而非计算机打交道，比如，我们构建的系统首先是给人使用的，其次才是让计算机理解。除此之外，业务的沟通探讨、技术方案的探讨等诸多事项都是人与人面对面的直接沟通交流，如果你不善于沟通，那么如何能够让别人明确你的用意，又如何顺利开展工作呢？

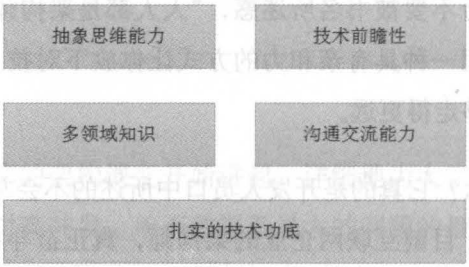


图 A-1　架构师的能力版图

最后，祝愿大家心想事成，工作顺利！

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

本书的核心内容如下：

□**分布式服务案例**：重点为大家演示了如何在阿里开源的服务治理框架Dubbo的基础上构建一个分布式调用跟踪系统，以一种可视化的方式来展现跟踪到每个请求的完整调用链，以及收集调用链上每个服务的执行耗时、整合孤立日志等。

□**大流量限流/消峰案例**：重点为大家演示了如何通过技术层面和业务层面两个维度来对流量合理实施管制，避免大促场景下因峰值流量过大而对系统造成较大冲击产生雪崩现象。

□**分布式配置管理服务案例**：重点为大家演示了如何基于ZooKeeper构建一个分布式配置管理平台，以及使用淘宝Diamond和百度Disconf系统来实现分布式配置管理服务。

□**大促场景下热点数据的读/写优化案例**：热点数据的读/写操作其实是秒杀、限时抢购场景下核心的技术难题。在大促场景下，由于峰值流量较大，大量针对同一热卖商品的并发读/写操作一定会导致后端的存储系统产生性能瓶颈，重点为大家讲解了大促场景下应该如何对热点数据的读/写操作进行优化。

□**数据库分库分表案例**：重点为大家演示了如何使用分库分表中间件Shark来帮助企业实施分库分表改造，以及分库分表后所带来一系列影响的解决方案。

上架建议：计算机 / 分布式系统

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：孙学瑛
责任编辑：徐津平
封面设计：李　玲

ISBN 978-7-121-31238-0

9 787121 312380

定价：69.00元